WL-TR-97-1194

AUTOMATED DESIGN OF BOARD AND MCM LEVEL DIGITAL SYSTEMS

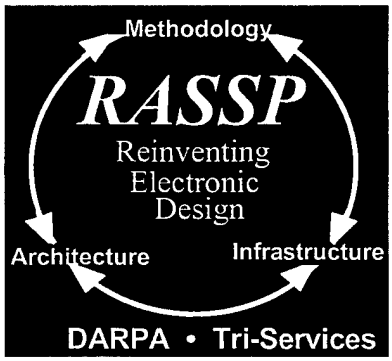
DR. RANGA VERMURI

DEPARTMENT OF ELECTRICAL
AND COMPUTER ENGINEERING
UNIVERSITY OF CINCINNATI
CINCINNATI OH 45221-0030

OCTOBER 1997

FINAL REPORT FOR 08/05/93-09/15/97

19980520 084



DTIC QUALITY INSPECTED 3

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AFB, OH 45433-7623

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility nor any obligation whatsoever. The fact that the Government may have formulated or in anyway supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

KERRY HILL, Project Engineer
Embedded Information Systems
Engineering Branch
AFRL/IFTA

JAMES S. WILLIAMSON, Chief
Embedded Information Systems
Engineering Branch
AFRL/IFTA

STANLEY E. WAGNER, Chief
Wright Site Coordinator
Information Directorate
AFRL/IFW

IF YOUR ADDRESS HAS CHANGED, IF YOU WISH TO BE REMOVED FROM OUR MAILING LIST, OR IF THE ADDRESSEE IS NO LONGER EMPLOYED BY YOUR ORGANIZATION, PLEASE NOTIFY AFRL/IFTA, WRIGHT-PATTERSON AFB OH 45433-7334 TO HELP US MAINTAIN A CURRENT MAILING LIST.

COPIES OF THIS REPORT SHOULD NOT BE RETURNED UNLESS RETURN IS REQUIRED BY SECURITY CONSIDERATIONS, CONTRACTUAL OBLIGATIONS, OR NOTICE ON A SPECIFIC DOCUMENT.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	10/15/97	Final 8/05/93-9/15/97

4. TITLE AND SUBTITLE
Automatic Design of Board and MCM Level Digital Systems

5. FUNDING NUMBERS
C: F33615-93-C-1316
PE 63739E
PR A268
TA 02
WU 07

6. AUTHORS
Dr. Ranga Vemuri

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Cincinnati
Department of Electrical & Computer Engineering and Computer Sciences
P.O. Box 210030
Cincinnati, OH 45221-0030

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Avionics Directorate
Wright Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7623
POC: Kerry L. Hill, AFRL/IFTA, 937-255-7698 x3604

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
WL-TR-97-1194

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for Public Release: Distribution is Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This is a Rapid Prototyping of Application Specific Signal Processors (RASSP) program funded by DARPA. The goal of this program is to develop languages, techniques, and tools for hardware, software cosynthesis of board- and MCM-level Digital Signal Processing (DSP) systems from very high level requirements specifications. A second goal is to develop a usage guide for the Level 2 Waveform and Vector Exchange Specification (WAVES) language. The program includes the development of, (1) VSPEC, a declarative interface requirements specification language for VHDL entities, (2) hardware/software cosynthesis techniques for embedded DSP systems, (3) hierarchical multi-technology hardware partitioning tools, (4) software compilation techniques to compile behavioral VHDL into C, (5) a WAVES Level 2 usage guide and (6) exploring WAVES usage in conjunction with BSDL and for hierarchical testing.

DTIC QUALITY INSPECTED 2

14. SUBJECT TERMS
VHDL, Design, Board-Level, MCM-Level, Specification Language

15. NUMBER OF PAGES
306

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

Table of Contents

1 Project Goals 1

2 Accomplishments 2

Attachments 4

Appendix A: Board and MCM Level Synthesis for Embedded Systems: The COMET Cosynthesis Environment 4

Appendix B: VSPEC: A Declarative Requirements Specification Language for VHDL 14

Appendix C: Pipelined Scheduling of Hardware-Software Codesigns 48

Appendix D: RECOD: A Retiming Heuristic to Optimize Resource and Memory Utilization in HW/SW Codesigns 57

Appendix E: Hardware/Software CoSynthesis: Multiple Constraint Satisfaction and Component Retrieval 71

Appendix F: A Retiming Based Relaxation Heuristic for Resource-Constrained Loop Pipelining 78

Appendix G: Multicomponent Partitioning for VLSI System Synthesis 93

Appendix H: Performance Modeling and Tradeoff Analysis During Rapid Prototyping 128

Appendix I: Performance Verification Using Partial Evaluation and Interval Analysis 143

Appendix J: Hierarchical Behavior Partitioning for Multicomponent Synthesis 159

Appendix K: Resource Constrained RTL Partitioning for Synthesis of Multi-FPGA Designs 169

Appendix L: Using Declarative Specifications and Case-Based Planning for System Synthesis 185

Appendix M: Extending VHDL to the Systems Level 223

Appendix N: Representing Abstract Architectures with Axiomatic Specifications and Activation Conditions 232

Appendix O: Formal Representations for Abstract System Evaluation 240

Appendix P: Abstract Architecture Representation Using VSPEC 246

Appendix Q: VSPEC: A Declarative Specification Methodology for System Synthesis 281

Appendix R: VSPEC: A Declarative Specification Methodology for System Requirements 288

Appendix S: Application of Software Synthesis Techniques to Composite Systems 295

1 Project Goals:

The Cosynthesis at Board and MCM Levels for Digital Signal Processors (COMET) Project is a RASSP Technology Base Program at the University of Cincinnati. RASSP (Rapid Prototyping of Application Specific Signal Processors) is an Advanced Research Projects Agency, Electronic Systems Technology Office (ARPA/ESTO) program. The COMET project is monitored by the US Air Force Wright Laboratory under contract number F33615-93-C-1316.

The goal of the COMET project is to develop languages, techniques and tools for hardware, software cosynthesis of board- and MCM-level Digital Signal Processing (DSP) systems from very high level requirements specifications. A second goal is to develop a usage guide for the Level 2 Waveform and Vector Exchange Specification (WAVES) language. The COMET project includes the development of, (1) VSPEC, a declarative interface requirements specification language for VHDL entities, (2) hardware/software cosynthesis techniques for embedded DSP systems, (3) hierarchical multi-technology hardware partitioning tools, (4) software compilation techniques to compile behavioral VHDL into C, (5) a WAVES Level 2 usage guide and (6) exploring WAVES usage in conjunction with BSDL and for hierarchical testing.

COMET project statement of work is as follows:

1. Extend VHDL to create VSPEC Specification Language (Requirement 3.2)
2. Develop technology driven VSPEC partitioner (Requirement 3.3)
3. Develop VSPEC-Embedded software Translator (Requirement 3.4)
4. Integration and distribution (Requirement 3.5)
5. WAVES usage guide for electronic module design development (Requirement 3.6)

2 Accomplishments

The accomplishments of the COMET project are summarized as follows:

1. VSPEC Development (CDRL A007)

   VSPEC as developed under this effort is a Larch interface language for VHDL. VSPEC provides a declarative specification mechanism for defining: (i) axiomatic requirements, (ii) activation conditions (iii) internal state, (iv) constraints, and (v) abstract architectures for systems. VSPEC is fully compatible with VHDL and provides requirement definitions for the interfaces of entities, functions and procedures.

   With the language definition complete a formal semantics for VSPEC was defined using the Larch Shared Language (LSL). This formal semantics is used to precisely define what VSPEC means and for verification. The VSPEC parser is being extended currently to generate LSL directly for use in verification tools.

2. VSPEC Partitioner (CDRL A008)

   Several partitioning approaches were developed under this project. Notable of these were the REBOUND tool and the genetic partitioner for codesigns.

   The REBOUND tool generates structural architectures. Accordingly, in the current version of the hardware/software partitioning tool, concurrent statements are limited to components. The approach is, however, extensible to other concurrent statements such as processes and blocks as well.

   The genetic partitioner contemplates hardware software codesigns based on a relaxation-based retiming strategy. The partitioner explores a large number of hardware alternatives and hardware/software bindings. To aid this process, a detailed performance estimator for pipelined and nonpipelined codesigns has been developed.

3. VSPEC-Embedded Software Translator

   Two tools for software synthesis were developed as a part of this effort. The first was a stand-alone parser developed around an ad hoc VHDL front end. This system generated code for the Texas Instruments TMS320 series DSP processor. Example systems included: (i) a compander system, (ii) an FFT subsystem, (iii) an IFFT subsystem, and (iv) an IIR filter. Each example was coded in VHDL-S, synthesized into C and evaluated on a TMS320 prototyping system.

   The examples synthesized generated the capability to generate C for the VHDL-S subset. Further, the initial example set demonstrated the ability to generate: (i) a generic operating system kernel, and (ii) interface routines to support executing the C code. VHDL is inherently parallel in nature while C is inherently sequential. Each VHDL-S process is transformed into a C process by the translator. These processes are managed by the simple operating system using message passing for interprocess communication. C routines are also generated to manage interfaces between software and associated hardware devices. This is primarily used for I/O associated with the DSP processor.

   The second software translation system took the initial results from the stand alone parser and incorporated them into the SAVANT environment. The SAVANT environment provides a much richer and more stable platform for building the translator. The object model was

extended to include C publishing routines and additional enhancements were added. The most significant addition was the ability to generate generic C from VHDL-S. The generic C code is standard C with TMS320-specific additions. The this code was tested on both Solaris and Linux platforms.

The final translator delivered here can generate code for either the TMS320 or a generic C system. It is based on the SAVANT toolset, but has not been ported to the most current SAVANT release.

4. Integration and Distribution

   All VSPEC software has been integrated and transferred to VHDL community by publications, presentations and repository access. The software can be accessed by anonymous File Transfer Protocol (FTP) by contacting the PI of this project. Several publications resulting from this project are appended in this report.

5. WAVES Usage Guides (CDRL A009)

   A WAVES Level-2 usage guide has been developed. In addition, two detailed case studies illustrating the use of WAVES Level-2 have been developed. A document describing the use of WAVES for testing boundary scan devices has been developed. A final document has been written describing the use of WAVES in conjunction with BSDL.

APPENDIX A :

Board and MCM Level Synthesis for Embedded Systems: The COMET Cosynthesis Environment *

Ranga Vemuri, Hal Carter and Perry Alexander
Department of Electrical and Computer Engineering
University of Cincinnati, ML. 30
Cincinnati, Ohio 45221-0030
Ph: 513-556-4784; Email: ranga.vemuri@uc.edu

Abstract

COMET *is a cosynthesis environment for application-specific electronic signal processing modules.* COMET *is capable of automatically synthesizing multicomponent hardware-software systems at the board- and* MCM- *levels. In the* COMET *environment, system-level specifications are written in* VSPEC, *a declarative annotation language for* VHDL *entities.* COMET *contains various* VHDL*-centered tools for hardware-software partitioning,* MCM *synthesis,* ASIC *synthesis, software compilation and performance analysis, and various* WAVES*-centered tools for board,* MCM- *and* ASIC *level testing.*

1 Overview

COMET (Cosynthesis at Board and MCM Levels for Digital Signal Processors) is a hardware-software cosynthesis environment for embedded signal processing modules. COMET users can synthesize single board application-specific DSP (digital signal processing) architectures. These target architectures, illustrated in Figure 1, can contain application-specific ASICs, FPGAs, MCMs, and off-the-shelf hardware components along with an off-the-shelf processor which executes applications-specific software as well as other kernel functions.

The users' view of COMET is shown in Figure 2. In a typical top-down design process, COMET users begin by writing a specification of the system's functional requirements and constraints in VSPEC. Then, using the hardware-software partitioning tool, a top-level hardware-software architecture is generated. The partitioning tool uses a library of reusable components. Each component is a DSP algorithm bound or to be bound to hardware or software. The component library also contains a set of off-the-shelf processors. The output of the partitioning tool is an architecture of hardware and software components whose behavior is specified in procedural VHDL. In addition, the software components are bound to an off-the-shelf processor and the hardware components are bound to various ASIC and packaging technologies. Hardware components in the design are submitted to hardware synthesis tools and the software components to software synthesis tools. An interface synthesis tool is used to synthesize all the interface logic to support inter-component hardware-software communication protocols. An architecture integration tool composes the various components into a coherent board-level design that can be processed by commercial board-level place and route tools.

COMET environment also contains test generation tools based on WAVES and performance analysis tools. COMET tools are also interfaced to various commercial and university tools, especially from the RASSP community, to facilitate simulation, logic synthesis, synthesis of board-level glue logic and ASIC, MCM and board-level layout synthesis.

2 VSPEC Specification Language

VSPEC is a declarative annotation language for VHDL entities. Through VSPEC designers specify requirements the system design should meet and constraints on its implementation. A VSPEC specification consists of a collection of logical statements and declarations

*The research reported in this paper is being conducted at the University of Cincinnati and is supported in part by by the ARPA RASSP program monitored by the Wright Lab, US Air Force under contract no. F33615-93-C-1316 and by the Solid State Electronics Directorate of the Wright Laboratory of the US Air Force under contract number F33615-91-C-1811.

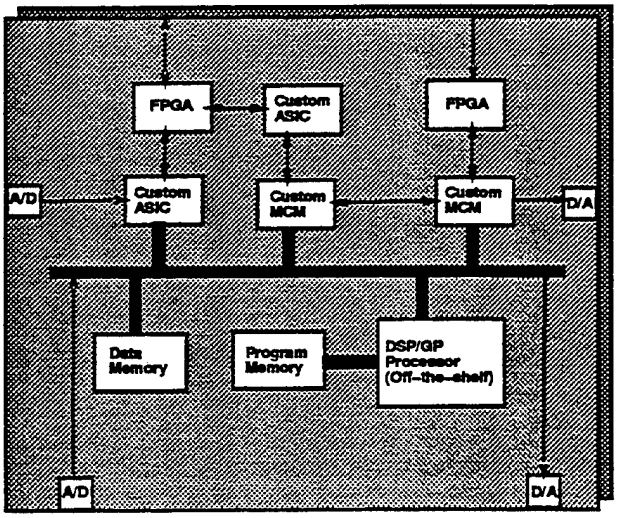


Figure 1: **COMET's Target Architectures**

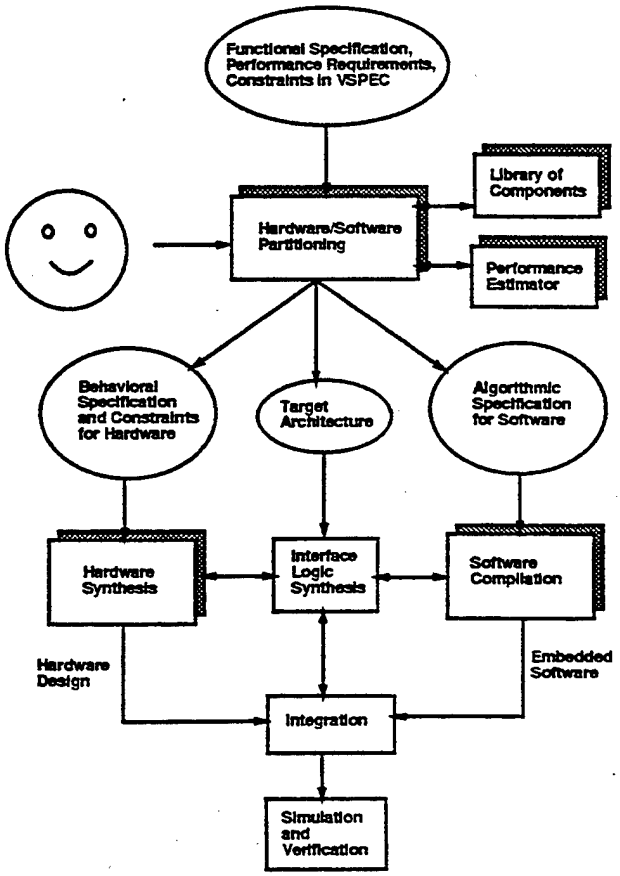


Figure 2: **COMET Cosynthesis Environment**

that annotate a VHDL `entity` construct. Consider the following `entity` specification of a multiplexor:

```
entity mux is
   port ( d0, d1, cntrl : in bit;
          output : out bit );
end mux;
```

In this example, the `entity` names the device and defines input and output ports. However, there is no indication of how the multiplexor functions or what performance constraints it must adhere to. A VHDL `architecture` describes the behavior or structure of an `entity`. Behavior can be described through communicating and concurrently executing sequential processes. Structure can be described through component instantiation and interconnection. VHDL allows the user to specify the behavior of a system by defining a single artifact (`architecture`) embodying that behavior. Although alternative behaviors may be specified by multiple `architectures` of the same `entity`, these `architectures` must be explicitly enumerated. Therefore, implementational biases occur while formulating the functional requirements since the user is forced to commit to one or more designs.

The VSPEC language was developed to support the definition of requirements prior to the specification of designs. VSPEC has constructs to allow its users to declaratively specify input pre-conditions, output post-conditions, state variables, constraints, and other requirements at the `entity` level. The following is a VSPEC definition of a simple multiplexor:

```
entity mux is
   port ( d0, d1, cntrl : in bit;
          output : out bit );
   ensures
     output = ((d0 and cntrl) or
               (d1 and not cntrl));
   constrained by
     power < 4 and
     size < 20
end mux;
```

This VSPEC entity describes the interface to the component as well as the desired function and constraints. The `ensures` clause declaratively states the function of the multiplexor. This definition allows many different implementations to be developed for this specification as long as the specification meets the requirement stated here. The `constrained by` clause specifies constraints placed on the *power* and *area* of the entity.

The VSPEC interface language affects only the VHDL `entity` declaration. Six VSPEC clauses are allowed in the entity:

- `assumes` *logical_expression*;

  Describes the pre-conditions that must be met before this entity can be used. The *logical_expression* is defined over the set of inputs of the device.

- `ensures` *logical_expression*;

  Describes post-conditions that must be true when the entity functions correctly. The *logical_expression* is defined as a relation between the inputs of the device and its outputs.

- `constrained by` *logical_expression*;

  Describes the constraints placed upon the entity. These constraints include size, timing, heat dissipation, power consumption and clock speed. The *logical_expression* is defined over pre-defined variables representing potential constraints.

- `state` *typed_identifier_list*;

  A list of typed variables used to store the state of the entity. These variables maintain their values from one `entity` invocation to the next.

- `modifies` *identifier_list* ;

  List of variables and signals this entity can modify. All elements listed must be defined in the `state` clause or in the `entity`'s `port` declaration and of type `out`, `inout`, or `buffer`.

- *VHDL_type* `based on` *logical_expression*;

  Associates a user defined VHDL type with a formal, logical definition. This allows inferences involving user defined types.

Architectures in VSPEC A general architecture is a collection of interconnected high level specifications that serves as a template for system definition. The general requirements of each component are known, the interaction between them is known, but the specifics of the implementation may not be known. The VHDL `architecture` construct supports specification of interconnections among entities. Whether the `entity` structures referenced by the `architecture` have associated architectures determines whether there are just requirements or designs associated with each `entity`.

Figure 3 shows a specification of a batch sequential sort and search system. The `entity` structures associated with each component in `batch-seq` are specified using VSPEC with no specific associated algorithm. The sort component must produce a sorted output and the search component must find a key given a sorted input. Algorithms for each, perhaps in the form of behavioral architectures, must be specified at a later time.

VSPEC Support Environment All VSPEC expressions translate into REFINE declarations. These declarations support a formal inference process, executable specifications and REFINE based software synthesis tools. REFINE is a language that allows programmers to write code in a wide range of styles. This includes high level constructs such as sets and transformation rules down to more traditional procedural language constructs such as loops and `if-then` statements [1]. REFINE specifications are executable. This allows designers to test their system at a very early point in the design process.

3 System Performance Estimation

Accurate performance estimation is critical to the success of a design synthesis system. The COMET performance estimator is used to evaluate the performance, in terms of area, speed, throughput rate, and power dissipation, of the library components as well as the performance of a contemplated hardware-software architecture of a system. The estimator can be used interactively or through the partitioning engine to filter inferior architectures and to select a constraint-satisfying hardware-software binding for a given specification. As shown in Figure 4, various hardware-software alternatives can be selected for each component in the architecture and for each selected configurations performance envelopes can be generated.

Hardware Performance Estimation: Performance estimation for hardware components is done by detailed analysis of the operational behavior of the component. A data-flow control-flow graph (DFG) is extracted from the behavioral specification. The DFG is scheduled across control-steps using register level hardware modules selected from a module library. From this scheduled and operator-bound DFG accurate estimates of area, clock-speed and throughput rate are made. Estimation of power consumption

```
entity example is
  port (list: in array of element;
        k: in integer;
        output: out element);
  modifies output;
  ensures
    (fa e:element)
      (output = e) <=>
        (e in input and
         k = key(e));
end example;

architecture batch-seq of example is
  component sorter is
    sort
      port (inlist: in array of element;
            outlist: out array of element);
  component searcher is
    bin_search
      port (inlist: in array of element;
            value: in integer;
            return: out element);
begin
  b1: sorter
        port map (list,tmp);
  b2: searcher
        port map (tmp,k,output);
end batch-seq;

entity sort is
  port (input: in array of element;
        output: out array of element);
  modifies output;
  ensures
    bag(output) = bag(input) and
    ordered(output);
end sort;

entity bin_search is
  port (input: in array of element;
        k: in integer;
        output: out element);
  modifies output;
  assumes
    ordered(input);
  ensures
    (fa e:element)
      (output = e) <=>
        (e in input and
         k = key(e));
end bin_search;
```

Figure 3: Batch sequential architecture for finding a value in a list.

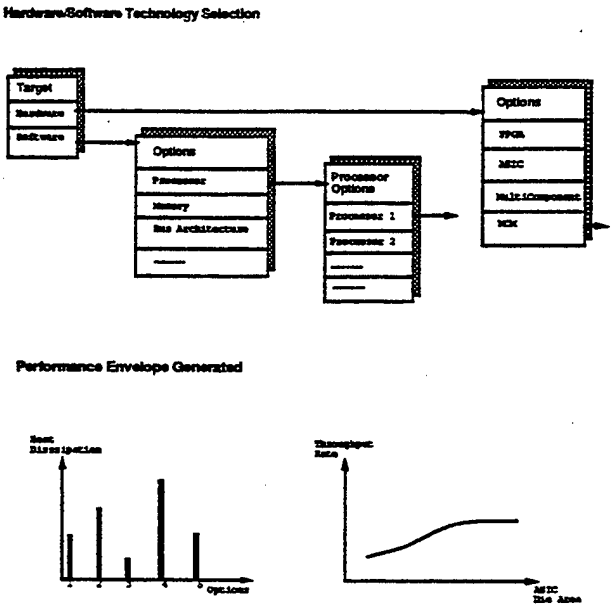


Figure 4: **Performance Estimation**

is based on the generation of profile data for typical stimuli of the component. The profile data is used to generate estimates of switching activity in the final design. Data from a technology library that contains both ASIC fabrication and packaging technology profiles is used to generate concrete technology-dependent estimates from the abstract estimates. Some of the hardware performance estimation work has been done as part of the MSS and DSS projects [3, 2].

Software Performance Estimation A static performance evaluation method based on ISA and code models is being developed to provide estimates of DSP software execution time. These estimates will be used to guide system and software partitioning such that timing constraints can be satisfied by the software synthesis algorithms. Once software has been created and compiled, the machine code is evaluated to assess whether timing constraints and throughput requirements have been satisfied. The static performance evaluation method consists of two graph theoretic models: (1) a pipelined instruction execution time (PIET) model which is accurate to the clock cycle level, and (2) an instruction stream execution graph (ISEG) model. The PIET model is constructed for each processor with a unique instruction set architecture and takes into account all data path dependencies including inter-instruction dependencies for accu-

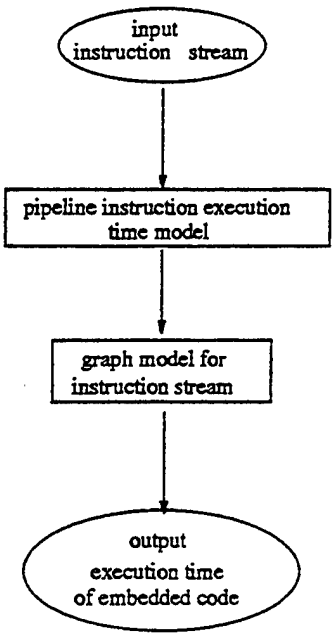


Figure 5: **Software Performance Evaluation**

rate time evaluation. The ISEG model is constructed for each software program being analyzed and is directly generated from the machine language instruction stream. The ISEG model evaluates all data and control paths within the instruction stream during its formation.

The flow of activities to perform static performance evaluation is shown in Figure 5. The objective is to obtain the estimated time of execution between any two points in the instruction stream. This time is obtained as an aggregate of the individual operation times of each instruction in the instruction stream given the PIET model of the ISA of the target processor. All pipelined activity and potential hazards are considered. The execution of each successive sequential instruction is evaluated until a branch instruction is seen. These successive sequential instructions are grouped into into basic blocks. The number of machine cycles for each basic block is determined using the PIET graph. The ISEG graph is created as a standard control flow graph where basic blocks and branch instructions are represented as nodes in the graph. Edges in the graph represent flow of control.

The determination of execution time between any two nodes in the graph proceeds by iteratively reducing the flow network between the two nodes until the two nodes are merged into a single node. Each reduction step proceeds by first examining the flow network and identifying a basic structure which can be reduced, followed by computing the execution time of basic structure and reducing the structure to a single node whose label is the derived execution time. Branches and loops are assessed based on the branch taken/not taken probabilities which are in turn obtained from the benchmark data patterns at the inputs of the software being evaluated. Note that this data is usually expressed in worst-case terms if worst-case execution performance estimates of the software is desired. If the estimated execution time of the entire software program is desired, the entire ISEG graph is reduced to one node by the graph analysis algorithm. The estimated execution time can then be compared with the timing constraints of the system to determine if the synthesized software satisfies the performance goals.

Reusual Behavioral Components COMET uses a library of reusable hardware, software or unbound components for synthesis. Performance of each library component is characterized using the same performance estimation tools described above. System synthesis in COMET is dependent on the availability of one or more library components for each function specified in VSPEC. If a VSPEC function in a specification has no corresponding component in the library, then the user is asked to supply a component along with its operational behavior description in VHDL. The performance of the description for various target hardware and software technologies will be evaluated using the performance estimation and the component along with this data will be stored in the library for later use (Figure 6).

4 System Partitioning

The goal of system partitioning is to generate a first level hardware-software architecture of the system by partitioning the system specification into specifications of hardware components and software components. The hardware components will be further processed by hardware synthesis tools. The software components will be bound to execute on a selected DSP or general purpose processor configuration. The hardware and software components will be connected to constitute a VSPEC-VHDL architectural description of the system. The functional requirements and constraints stated in the VSPEC specification drive the derivation of the specific hardware-software mix. Fig-

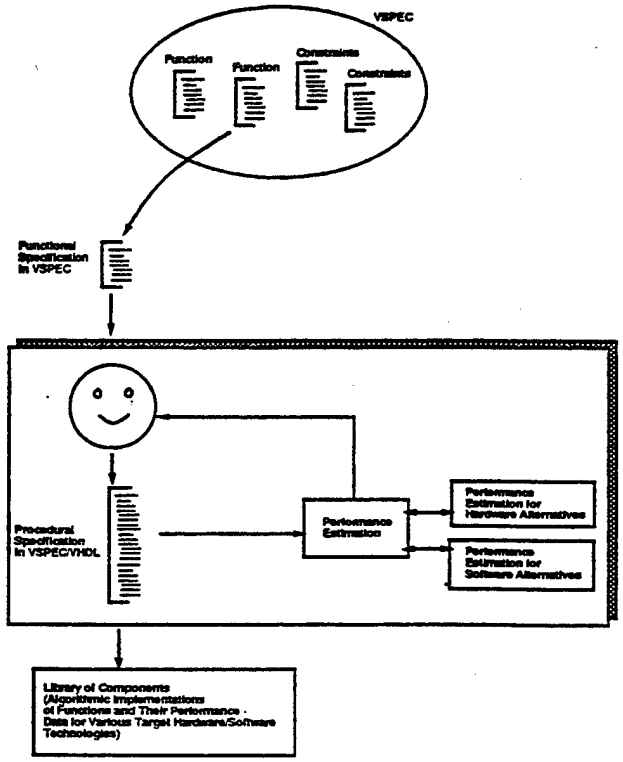


Figure 6: **Performance Analysis for Library Components**

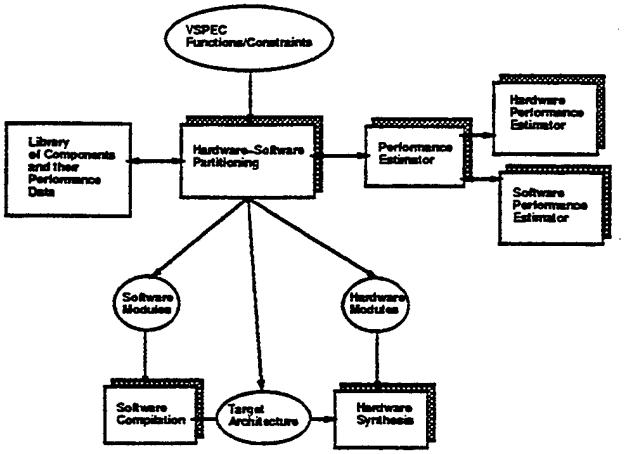


Figure 7: **System Partitioning in COMET**

ure 7 shows the system partitioning tool in COMET.

Initially, the VSPEC system specification is refined based on queries into the design library. As a result of the queries, components are selected based on their ability to satisfy the system function and constraint attributes. In case the existing components do need meet the requirements, a design that partially satisfies the requirements may be generated. Alternatively, the designer may be queried for additional components.

5 Hardware Synthesis

COMET hardware synthesis system consists of a multi-component partitioning engine and a set of synthesis tools for ASIC, FPGA and MCM synthesis (Figure 8).

Multicomponent Partitioning Engine The partitioning engine is a hierarchical partitioning and package binding tool that accepts behavioral specifications in VHDL, constraints on area, power consumption, pin counts, speed and cost and generates a hierarchical partition of the specification with each component in the partition bound to a package among a set of available packages. The partitioning engine uses a back-tracking algorithm for constraint-directed search. Power estimation is based on data gathered by dynamic profiling of the VHDL specification using typical stimuli.

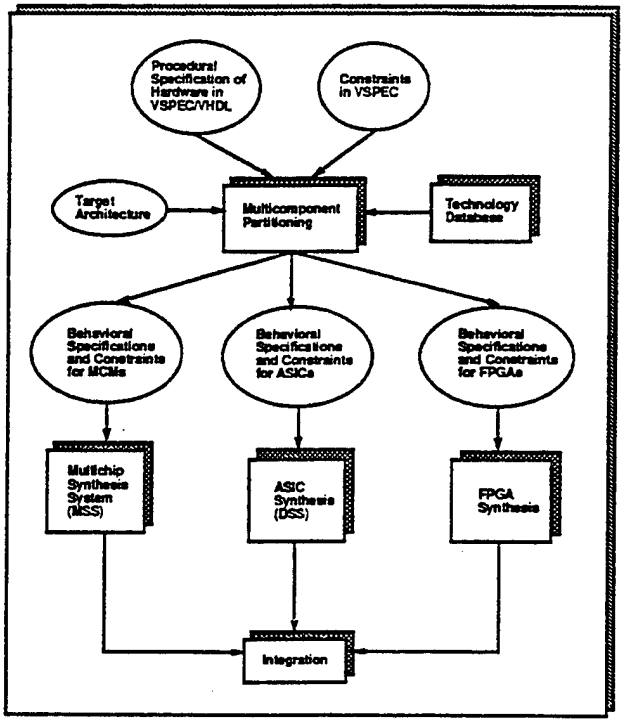


Figure 8: **Hardware Synthesis Flow in COMET**

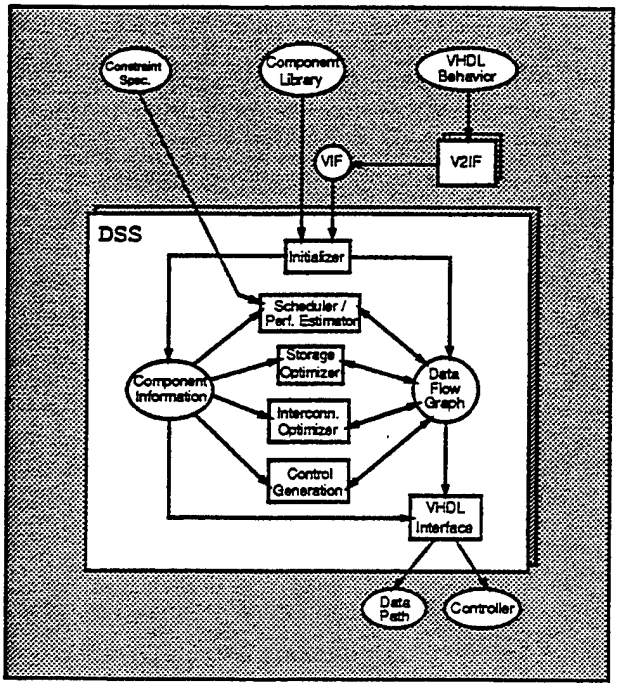


Figure 9: **DSS High Level Synthesis System**

High Level Synthesis of ASICs: DSS The ASIC synthesis system DSS (Distributed Synthesis System) accepts behavioral specifications in VHDL and constraints on clock period and area. It generates register level designs in VHDL. Register level designs contain two parts: a data path and a finite state controller. The data path is in the form of structural VHDL in which each component is instantiated from a predefined parameterized register level component library. DSS architecture is shown in Figure 9. For an overview of the DSS system, see [2].

Register level designs generated by DSS can be processed using various layout synthesis tools including Lager IV and Mentor Graphics' GDT tools. Figure 10 shows design flow using the DSS system. Test vectors for register-level and switch-level simulations are automatically created using a test-bench compiler. Figure 11 shows the design a processor (Move Machine) generated by DSS. DSS has been used to generate numerous designs including some industrial strength designs by Texas Instruments [4].

MCM Synthesis: MSS MCM synthesis environment MSS [3] is embedded in COMET to facilitate syn-

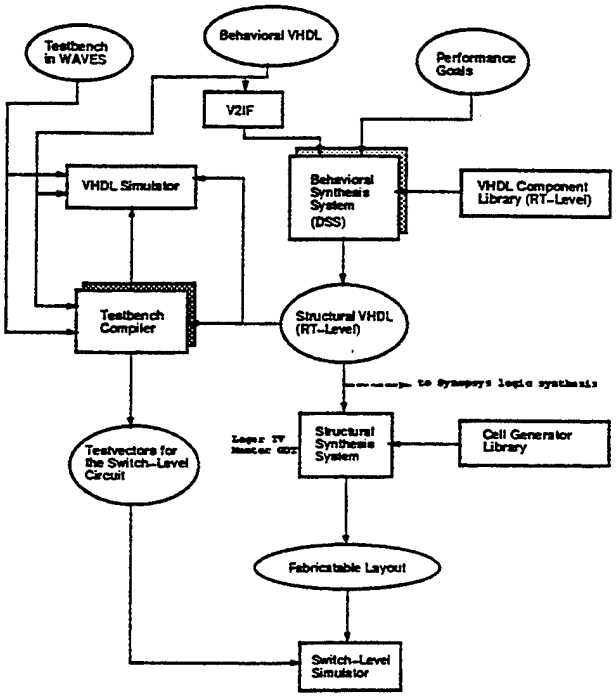


Figure 10: **ASIC Synthesis Using DSS**

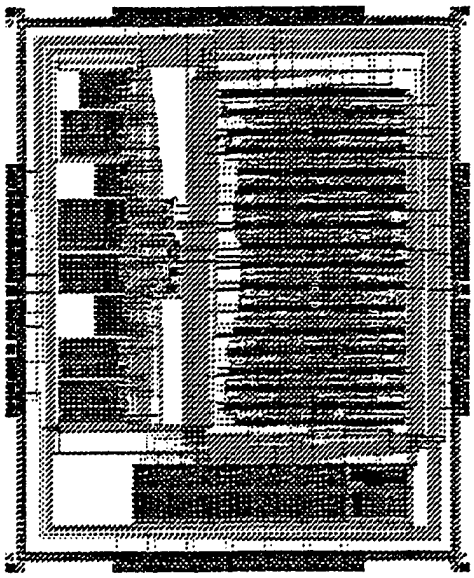


Figure 11: **Move Machine**

thesis of multichip modules. The tools in the MSS environment are shown in Figure 12. Behavior specifications for MSS are written in VHDL. Performance descriptions are written in PDL (Performance Description Language) [5, 6]. Multichip designs can be generated in two ways. As shown in Figure 12, register level designs generated by DSS can be partitioned into multiple chip designs. Alternatively, as shown in Figure 13 an integrated behavior synthesis and partitioning step can be performed to obtain multichip designs directly. These multichip designs are then processed by package level place and route tools. We currently use Mentor Graphics MCM Station and plan to use Harris EDA Finnesse system in near future. Figure 14 shows the MCM design of the Find processor generated using the MSS tools.

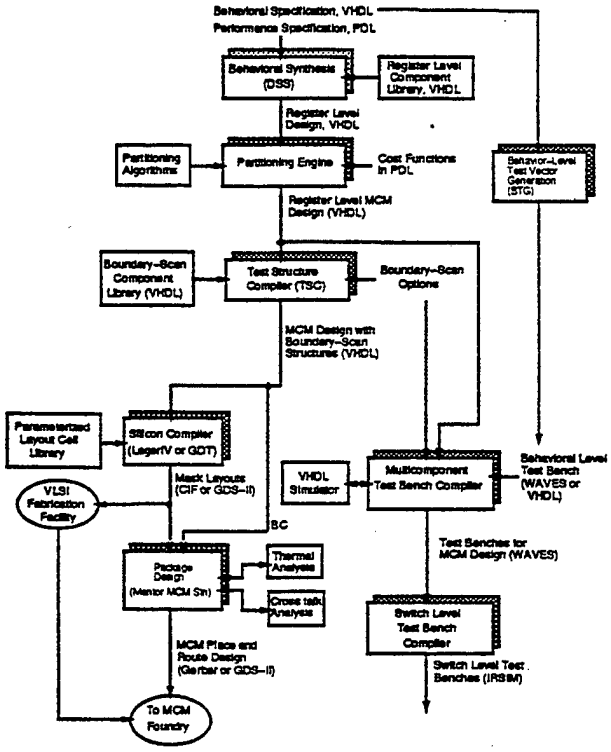


Figure 12: **Multicomponent Synthesis System, MSS**

6 Software Synthesis

The software synthesis tools in COMET translates DSP-based software behavioral specifications expressed in a subset of VHDLinto efficient machine code capable of being executed in a multiprocessor environment. The overall approach to software synthesis, shown in Figure 15 is to translate behavioral descriptions expressed in VHDL into C and then use commercial C compilers to translate C into machine code to execute on the target processor. In this way, any processor with a C compiler can be used as a target. The currently supported processors are the Texas Instruments

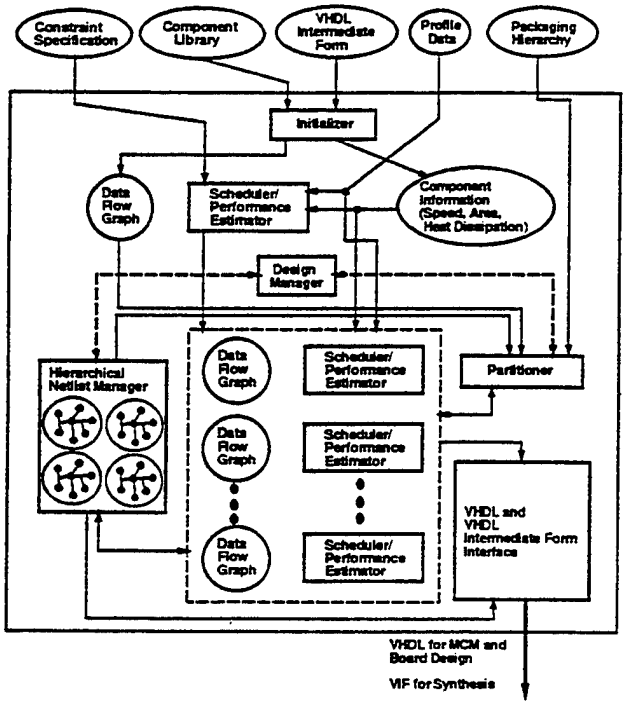


Figure 13: **Partitioning with Synthesis in MSS**

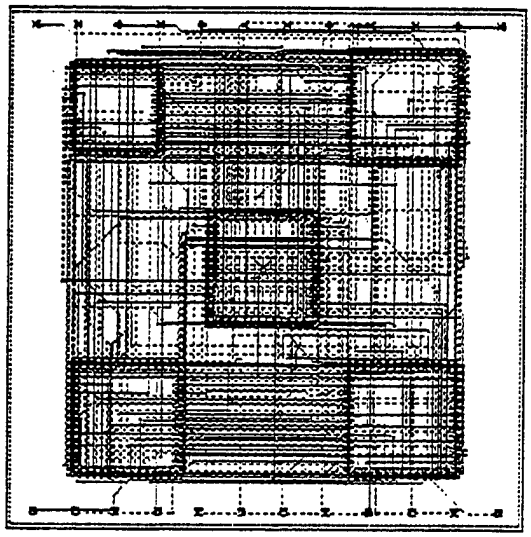


Figure 14: **Find MCM**

TMS430C51, Sun Microsystems SPARC, and Intel 80386. As explained previously, the compiled code can be statically analyzed for timing performance to ensure compliance with timing constraints expressed in the VSPEC specification.

The VHDL subset used as input for software synthesis is similar to that used for ASIC synthesis [2]. VHDL behavioral constructs are fully supported along with a limited subset of structural constructs. Explicit timing , such as VHDL `after` clauses or specific time in *wait* statements, is not supported.

Translation into C is a straightforward process. The code generator is encapsulated in template functions to allow future extensions to languages other than C. For example, the code generator objects can be easily changed to output Ada source code strings rather than C source code strings.

The execution environment consists primarily of a small multitasking operating system kernel which will provide interprocess communication service, task management, and input/output support. The task scheduler will create, maintain and monitor all tasks in the run-time space, while the interprocess communication protocol will support a simple message pass-

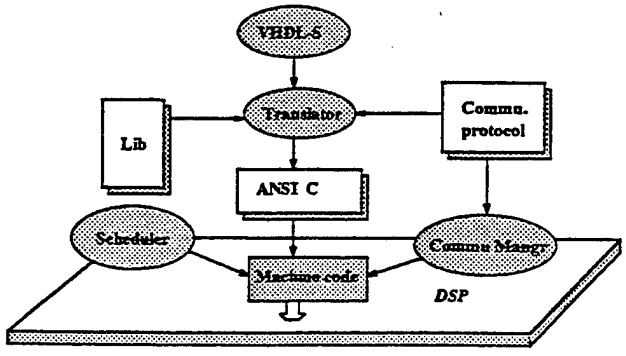


Figure 15: Software Synthesis Flow in COMET

ing mechanism where a process writes its request and data in a message channel whenever it tries to communicate with others, and then optionally waits until a response is received. The I/O drivers will provide a simple stream capability with support for objects of arbitrary width.

7 Test Tools in COMET and MSS

COMET and MSS contain various tools for the testing and simulation of designs as the design process progresses. Designs from behavioral level to gate level are expressed in VHDL; any VHDL simulator can be used to simulate these designs. Test vectors are automatically generated at various levels of abstraction. These test generation tools take WAVES files as input and generate WAVES files as output. As shown in Figure 12, at the behavior level, the users write WAVES data sets to simulate behavioral descriptions. A multicomponent test-bench compiler translates data sets into individual WAVES data sets for each of the chips in the multichip design. The data set also contains expected responses so that automatic comparison between expected and actual responses can take place. Switch level simulation is facilitated by a switch-level test-bench compiler that generates switch-level tests from WAVES data sets.

In addition to the automatically generated tests, users can add additional tests to the WAVES data sets. To aid users in this process, WAVES usages guide for multicomponent designs addressing both WAVES Level 1 and Level 2 constructs are being developed [7, 8].

8 Conclusion

COMET design environment is under development as part of the RASSP program. MSS and DSS systems have been operational for over two years; their development has been funded separately by Solid State Electronics, Wright Lab and ARPA. COMET tools significantly advance the state of the art in automated and vertically integrated synthesis systems. Various tools in the COMET cosynthesis system are interfaced with other commercial and university tools within the RASSP community and produce design and test files in standard notations such as VHDL and WAVES. Through the use of the VSPEC notation, the COMET environment supports design synthesis from abstract, declarative specifications of board and MCM level digital signal processing architectures.

Acknowledgements Besides the authors, the COMET team includes Phil Baraona, Yueqin Lin, Rick Miller, John Penix, John Rowe, Vinoo Srinivasan, Jeff Walrath, and Danjin Wu.

References

[1] "Refine User's Guide, Version 3.0", Reasoning Systems Inc., Palo Alto, CA, May, 1990.

[2] Jay Roy, Rajiv Dutta, Nand Kumar, and Ranga Vemuri, "DSS: A Distributed Synthesis System for VHDL Specifications", Design and Test of Computers, pp. 18-32, June 1992.

[3] Ranga Vemuri et al., "An Integrated Multicomponent Synthesis System for MCMs", IEEE Computer, pp. 62-74, April 1993.

[4] Ranga Vemuri et al., "Experiences in Functional Validation of a High Level Synthesis System", *30th Design Automation Conference*, pp. 194-201, 1993.

[5] Ram Mandayam and Ranga Vemuri, "Performance Specification using Attribute Grammers", Design Automation Conference, June 1993.

[6] Ranga Vemuri, Ram Madayam, and Vijay Meduri, "Performance Estimation and Tradeoff Analysis During Rapid Prototyping", University of Cincinnati, July 1994.

[7] W. Zhou, H. Hirsch, and R. Vemuri, "WAVES and VHDL Modeling Guidelines", RL-TR-94-56, Rome Laboratory, May 1994.

[8] Jeff Walrath, John Rowe and Ranga Vemuri, *WAVES Level 2 Usage Guide with Annotated Examples*, under preparation.

APPENDIX B :

VSPEC: A Declarative Requirements Specification Language for VHDL

Phillip Baraona, John Penix and Perry Alexander
Department of Electrical and Computer Engineering
Knowledge-Based Software Engineering Lab
The University of Cincinnati
Cincinnati, OH USA 45221-0030
pbaraona@uceng.uc.edu

December 14, 1994

Abstract

VHDL allows a designer to describe a digital system by specifying a specific design artifact that implements the desired behavior of the system. However, the operational style used by VHDL forces the designer to make design decisions too early in the design process. In addition, there is no means for specifying non-functional performance constraints such as heat dissipation, propagation delay, clock speed, power consumption and layout area in standard VHDL. Thus, VHDL is not appropriate for high level requirements representation. VSPEC is a Larch-like requirements language used with VHDL that solves these problems. VSPEC adds seven clauses to the VHDL entity structure that allow a designer to declaratively describe the data transformation a digital system should perform and performance constraints the system must meet. The designer axiomatically specifies the transformation by defining predicates over entity ports and system state describing input precondition and output postconditions. A constraints section allows the user to specify timing, power, heat, clock speed and layout area constraints. In combination with the architecture declaration, collections of VSPEC specified components can define a high level architecture as interconnected collections of components where requirements of components are known (via a VSPEC description), but implementations are not. This work presents the VSPEC language and associated design methodology.

1 Introduction

VSPEC is a language for declaratively specifying digital systems. It annotates the hardware description language VHDL by adding seven new clauses to the entity construct. These clauses allow a digital system to be specified using a declarative style as opposed to the operational style of VHDL.

With VHDL alone, the only way to specify a digital system is by describing a specific design artifact that implements the system's desired behavior. On the other hand, VSPEC allows the designer to describe the function of the system without defining the eventual implementation. In short, VSPEC allows the specification of "what" a system should do as opposed to the VHDL description of "how" the system will do it. This is consistent with Hoare's definition of specifications [9].

In addition to allowing the specification of "what" instead of "how", VSPEC addresses another limitation of VHDL: specifying performance constraints. When designing a digital system, meeting certain non-functional (i.e. performance) constraints is equally as important as creating a system that functions properly. A flight control system so slow that it calculates a flight correction after the plane crashes is obviously inadequate. Since they are so important in digital systems, performance constraints should be specified very early in the design process. However, VHDL does not provide a consistent mechanism for specifying these types of constraints. VSPEC addresses this problem by allowing the designer to specify performance constraints such as heat dissipation, propogation delay, clock speed, power consumption and layout area.

Another way of viewing VSPEC is as a Larch style interface language for VHDL. The Larch family of specification languages supports a two-tiered, model-based approach to specifying programs [7]. A Larch specification consists of components written in two languages: an Interface Language and the Larch Shared Language. Interface languages are used to specify the interfaces between program components, including component inputs and outputs as well as the observable behavior of the component. Interface languages exist for a variety of programming languges, including C [6], C++ [2], Modula-3 [12] and Ada [5].

Definitions written in the Larch Shared Language (LSL) are the second component of a Larch specification. LSL is a formal algebraic language that defines the underlying sorts and operators used in the Larch Interface Languages [8, 3]. In the VSPEC system, REFINE [17] is the primary shared language. REFINE is a language that contains a wide range of constructs, from high-level

sets and transformations down to more traditional loops and conditional statements. All VSPEC clauses can be translated into a REFINE representation. There are two main reasons REFINE was chosen as the primary shared language for VSPEC.

First, LSL specifications are not executable. Since REFINE is a broad spectrum programming language, some VSPEC specifications are executable. This is a very important feature for a digital system specification language such as VSPEC. VHDL descriptions of digital system are simulated as early as possible in the design cycle so that bugs can be found when they are the least expensive to fix. This same concept extends to a VSPEC requirements specification of a system. The sooner a bug in the requirements specification is found, the less expensive it is to fix. One way that problems with the specification can be found at the earliest possible point in the design cycle is by executing the specification.

The second reason REFINE was chosen as the primary shared language is that it supports synthesis of behavioral VHDL from VSPEC. REFINE is one part of a suite of software synthesis tools. Supporting synthesis of behavioral VHDL from VSPEC is one of the main long term goals of this research.

VSPEC is one part of the COMET research project. The goal of this project is to develop better techniques for rapid prototyping of digital signal processing systems. A detailed description of COMET is beyond the scope of this paper [22], but as the project overview in Figure 1 shows, a COMET user begins by writing a description of the function and constraints of the system in VSPEC. This description is then used to partition the system into hardware and software components with an architecture for connecting these pieces together. Each of these components is synthesized and integerated into a board level implementation of the system that is simulated and verified against the original specification.

The remainder of this paper gives a more detailed description of VSPEC. The next section briefly describes the VHDL constructs that are important in VSPEC. Section 3 gives a detailed description of each of the seven VSPEC clauses. This is followed by an extended example where VSPEC is used to

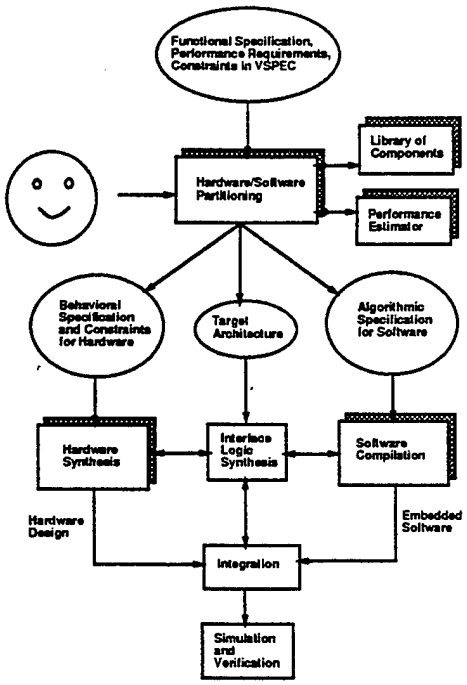


Figure 1: Overview of the COMET project.

specify a small microprocessor. Following this is a section that describes the formal representation of VSPEC. Section 6 discusses other work related to VSPEC and the paper concludes with a description of the current status and future directions for this research.

2 Important VHDL Constructs

This section gives a very brief description of two of the VHDL constructs used in VSPEC. It contains enough information to explain why the VSPEC annotations are needed in a specification language for digital systems. For a more complete description of VHDL, refer to the VHDL language reference manual [10] or a textbook on VHDL [16]. If you are already familar with VHDL, you can skip this section and begin reading about the VSPEC clauses described in Section 3.

Two of the more important constructs in VHDL are entities and architectures. A VHDL entity

declares a digital component by defining the component's interface. The function of the component is not defined in the entity structure. Instead, each entity has one or more associated architectures where the function of the component is described. This is the "big picture" of how entities and architectures are used. The next few paragraphs give a more detailed description of each of these constructs, starting with the syntax for a VHDL entity:

⟨*entity_declaration*⟩ ::= ENTITY ⟨*identifier*⟩ IS
⟨*entity_header*⟩
⟨*entity_declarative_part*⟩
[BEGIN
⟨*entity_statement_part*⟩]
END [⟨*Entity_simple_name*⟩];

The most important portion of the entity declaration is the entity header. The only part of the entity header currently used in VSPEC is the port clause. A port clause defines the inputs and outputs of the component. Here is an example entity declaration for a simple two input multiplexor:

```
ENTITY vhdl_mux IS
   PORT ( D0, D1, cntrl : IN BIT;
          output : OUT BIT );
END vhdl_mux;
```

Notice that this entity merely defines the types of the inputs and outputs to the multiplexor. It does not contain any description of the function of the entity.

The function of the multiplexor is described in the VHDL architecture. Each entity has one or more associated architectures. An architecture is used to define the behavior of a specific implementation of an entity. The syntax of the architecture construct is as follows:

⟨*architecture_body*⟩ ::= ARCHITECTURE ⟨*identifier*⟩ OF ⟨*Entity_name*⟩ IS
⟨*architecture_declarative_part*⟩

BEGIN
⟨*architecture_statement_part*⟩
END [⟨*Architecture_simple_name*⟩];

Detailed descriptions of each portion of the architecture are beyond the scope of this document (see [10, 16]). Suffice it to say that the declarative part of the architecture defines the types, signals and components used by the architecture while the statement part defines the behavior or structure of the entity. Consider the following architecture for the multiplexor entity above:

```
ARCHITECURE behavior OF vhdl_mux IS
   BEGIN
   PROCESS ( D0, D1, cntrl )
   BEGIN
      IF cntrl = 0 THEN output <= D0;
      ELSE output <= D1;
   END PROCESS
END behavior;
```

This is an example of a behavioral architecture. Behavioral architectures use ADA-like programming constructs to define the function of an entity. In this simple example, an if-then statement is used to assign a value (<= is used for signal assignment) to the output port based on the value of `cntrl`. Although this is a simple example, behavioral architectures can be quite complex. Auxiliary procedures and functions can be written in the declarative part of the architecture and entire packages of library routines can be used within the architecture. With these auxiliary procedures and packages, a behavioral architecture can be defined using a large program. No matter what size, all behavioral architectures have one thing in common: they define a single implementation of the behavior of an entity.

Structural architectures are the second common type of VHDL architectures. This architecture type defines the subcomponents an entity is composed of and how those subcomponents are connected. For example, the behavior of the multiplexor could also be defined using and, or and not gates connected as shown in Figure 2. In VHDL, this is represented using the following architecture:

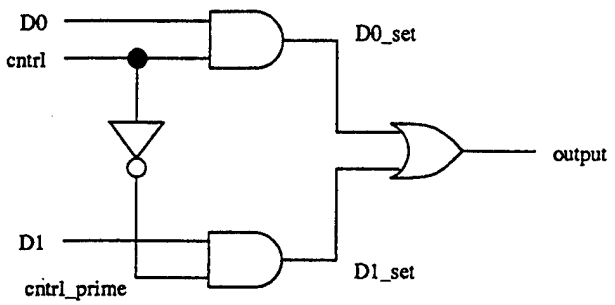


Figure 2: Structural Implementation of Multiplexor

```
ARCHITECTURE structure OF vhdl_mux IS
   COMPONENT and_gate PORT (in1, in2 : IN BIT; output : OUT BIT);
   END COMPONENT;
   COMPONENT or_gate PORT (in1, in2 : IN BIT; output : OUT BIT);
   END COMPONENT;
   COMPONENT not_gate PORT (input : IN BIT; output : OUT BIT);
   END COMPONENT;
   SIGNAL D0_set, D1_set, cntrl_prime : BIT;
BEGIN
   and_1 : and_gate PORT MAP (in1=>D0, in2=>cntrl, output=>D0_set);
   and_2 : and_gate PORT MAP (in1->D1, in2=>cntrl_prime, output=>D1_set);
   not_1 : not_gate PORT MAP (input=>cntrl, output=>cntrl_prime);
   or_1 : or_gate PORT MAP (in1=>D0_set, in2=>D1_set, output=>output);
END structure;
```

In this example, the declarative part of the architecture defines three components and three signals. The component declarations (`and_gate`, `or_gate` and `not_gate`) define the inputs and outputs of three sub-components that will be used in this architecture. The behavior and/or structure of these three sub-components must be defined by an entity/architecture pair somewhere else in the system (not shown here). Another VHDL construct, the configuration, is used to map components to the the entity/architecture pair that define the behavior of the component. The three signals declared (`D0_set`, `D1_set` and `cntrl_prime`) are used to connect these three components together as shown in Figure 2.

Instances of each of the components in the architecture's declarative part are created in the statement part (between begin and end). The port map for each instance shows how that particular

component instance is connected to the signals in the architecture.

Although this example is very small, the same basic concepts defined here scale to much larger systems. This multiplexor could be part of an ALU which is a sub-component of a large microprocessor which is itself one component on a board level system. The same type of structural architecture is used to connect the system together at each of these levels. The lowest level (the and, or and not gates in this example) contains a behavioral description of the components. Because VSPEC is an extension of VHDL, these features for dealing with large systems are also found in VSPEC.

3 The VSPEC Clauses

The VSPEC language annotates VHDL by adding seven new clauses to the entity structure. The modified syntax for the entity structure becomes:

⟨*entity_declaration*⟩ ::= ENTITY ⟨*identifier*⟩ IS
    ⟨*entity_header*⟩
    ⟨*vspec_clause_list*⟩
    ⟨*entity_declarative_part*⟩
    [BEGIN]
    END [⟨*Entity_simple_name*⟩];

The only change made to the VHDL syntax was the addition of the optional VSPEC clause list to the entity declaration. [1] All other constructs remain intact. A VSPEC clause list is a list of the seven VSPEC clauses separated by commas:

⟨*vspec_clause_list*⟩ ::= ⟨*vspec_clause*⟩ { ; ⟨*vspec_clause*⟩ } ;

[1] This statement is not completely accurate since VHDL's expression syntax was also extended to include quantifiers, logical implication and support for sets and sequences. This is described in a little bit more detail in the VSPEC Language Reference Manual [13].

⟨*vspec_clause*⟩ ::= [⟨*requires_clause*⟩] | [⟨*ensures_clause*⟩] | [⟨*state_clause*⟩] |
[⟨*constrained_by_clause*⟩] | [⟨*modifies_clause*⟩] | [⟨*based_on_clause*⟩] | [⟨*includes_clause*⟩]

These VSPEC clauses can be grouped into four broad classes. The first class defines the function of the entity and includes the `requires` and `ensures` clauses. The next class declares the internal state of the entity in the `state` clause. The third type of VSPEC clause is used to define the constraints placed on the system. The `constrained by` and `modifies` clauses fall into this category. Finally, the `includes` and `based on` clauses are used to help map the VSPEC definition to its formal representation in REFINE. These are the only two clauses that can appear more than once in a VSPEC clause list. The following sub-sections describe each of these clauses in a little bit more detail.

3.1 Requires Clause

⟨*requires_clause*⟩ ::= REQUIRES ⟨*logical_expression*⟩ ;

The `requires` clause states the pre-condition for the entity. If the entity's inputs and current state make the `requires` logical expression true, then the entity is guaranteed to perform its specified function. The behavior of the entity is undefined if the `requires` clause is false. A designer that uses an entity specified with VSPEC must ensure that the `requires` logical expression is true before the entity is used. Consider the following example:

```
ENTITY search IS
  PORT (input : IN ARRAY OF record_type;
        key : IN INTEGER;
        output : OUT record_type)
  REQUIRES sorted(input);
  ENSURES element_of(output, input) AND output.keyval = key;
  INCLUDES "sort.re", "set.re";
END search;
```

In this example, `sorted` is a function defined in the file "sort.re" (see description of `includes` clause in Section 3.6) that returns true if the array passed in is in order and false otherwise. The `search` entity above will only function properly if the input array is sorted. If the input is not in order, the function of `search` is undefined. The function of all entities is undefined if the `requires` clause is false. For this reason, it is best to keep the pre-conditions expressed in the `requires` clause as simple as possible. The more conditions that must be met for the requires clause to be true (i.e. the more complex the pre-condition), the more difficult it will be to meet the pre-condition and use the entity. Thus, the pre-condition should be kept as simple as possible. A pre-condition of true implies the entity has no pre-condition. It must function properly on all input values.

One portion of the `requires` clause definition has been kind of ignored to this point: What is a logical expression? All logical expressions in the VSPEC clauses use a syntax that is an extension of VHDL. The VHDL expression syntax supports the standard boolean expressions and, or and not. VSPEC extends this syntax by adding constructs for variable quantification and logical implication. In addition, the VSPEC expression syntax includes constructs for sets and sequences. See the VSPEC Language Reference Manual [13] for a more detailed description of the syntax of VSPEC expressions.

3.2 Ensures Clause

⟨*ensures_clause*⟩ ::= ENSURES ⟨*logical_expression*⟩ ;

The `ensures` clause states the post-condition of the entity. A designer implementing an entity specified with VSPEC must ensure that this logical expression is true whenever the entity processes valid input (i.e. input that makes the `requires` logical expression true). Consider the following example:

```
ENTITY vhdl_mux IS
   PORT ( D0, D1, cntrl : IN BIT;
```

```
        output : OUT BIT );
   REQUIRES true;
   ENSURES output = (D0 AND cntrl) OR (D1 AND (NOT cntrl));
END vhdl_mux;
```

This is a VSPEC description of the two input multiplexor specified in Section 2. The requires clause states that this entity is guaranteed to work for all legal values of the input varaibles. The logical expression in the ensures clause declaratively specifies the function of the entity. The logical expression is a condition that must be true when the entity functions properly. Thus, the ensures logical expression describes the functional requirements of the entity.

For this simple multiplexor example, the differences between a VHDL behavioral description and VSPEC may not seem that significant. For a more telling example, consider the specification of a sorting component. In VHDL, the simplest way to speicify a sorter is an entity with a behavioral architecture describing its function. This behavioral architecture would be an ADA-like description of a specific sorting algorithm such as bubble sort or quicksort. This forces the design of the component to a specific implementation at a very early stage in the design process. In reality, this behavioral architecture is a description of "how" the sorter should work, not "what" the sorter should do. It biases the implementation towards a specific design (i.e. a bubble sort or quicksort) and forces a designer to deal with unneccessary detail at a very early point in the design process.

On the other hand, a sorting component could be described in VSPEC like this:

```
ENTITY sorter IS
   PORT ( input : IN ARRAY OF INTEGER;
          output : OUT ARRAY OF INTEGER );
   REQUIRES true;
   ENSURES permutation(output, input) AND
           sorted(output);
   INCLUDES "sort.re";
END sorter;
```

In this example, permutation is a function (defined in "sort.re") that returns true if output contains all the same elements as input while sorted is the same function used in Section 3.1.

This code describes a sorting component as something that ensures `input` and `output` contain the same elements and that `output` is in order. Thus, the specification above describes the functional requirements of the sorter without describing an implementation of a sorting algorithm. In other words, this definition describes "what" the sorter must do instead of defining "how" it should be done. VHDL alone does not allow this type of description. The VSPEC `ensures` and `requires` clause add this feature to VHDL.

3.3 State Clause

⟨*state_clause*⟩ ::= STATE ⟨*vspec_variable_declaration_list*⟩ ;

The purpose of the `state` clause is to define a list of variables that store the state of an entity. In most algebraic specification languages (such as Larch [7]), a computational unit is defined as a transformation from inputs to outputs. This type of transformation is not adequate for specifying systems with VSPEC. Unlike typical subprograms, an entity's local storage is not re-initialized for each use of the entity. Buffers and registers retain their values from one use of the entity to the next. The `state` clause provides a means to model this. The variables declared in the `state` clause serve as the local storage for the entity. In addition, hardware designers very naturally think in terms of the state of a device and the `state` clause allows them to extend this thought process to the specification of the digital system.

The syntax for a VSPEC variable declaration list is:

⟨*vspec_variable_declaration_list*⟩ ::= ⟨*vspec_variable_declaration*⟩ {, ⟨*vspec_variable_declaration*⟩ }
⟨*vspec_variable_declaration*⟩ ::= ⟨*identifier_list*⟩ : ⟨*subtype_indication*⟩

An identifier list is a comma-separated list of identifiers while a subtype indication is the VHDL construct used to declare the type of a variable. In most cases, this is just an identifier that names

the type of the variable(s) declared, but refer to the VHDL documentation for a more complete description [10, 16].

3.4 Constrained By Clause

⟨*constrained_by_clause*⟩ ::= CONSTRAINED BY ⟨*logical_expression*⟩ ;

While the **ensures** clause is used to describe the functional requirements placed on a system, the **constrained by** clause is used to describe the performance requirements of the system. Consider the affect of adding the following clause to the sorter example in Section 3.2:

```
CONSTRAINED BY
   size <= 2 um * 5 um AND
   power <= 20 mV AND
   input<->output <= 100 us;
```

With this additional clause, the VSPEC entity now supplies information about the area the entity must be implemented in, the maximum power consumption of the entity and the pin to pin timing for the entity. VHDL does not provide a convenient way to specify these types of performance constraints. The **constrained by** clause provides a standard method for specifying the non-functional requirements of the system.

The logical expression used in the **constrained by** clause must be a conjunction of constraint expressions. The syntax for these expressions is:

⟨*constraint_expression*⟩ ::= ⟨*constraint_type*⟩ ⟨*relational_op*⟩ ⟨*constraint_value*⟩

where the relational operators are the standard VHDL operators <=, <, >=, >, = and /= (not equal) and the constraint value is either a physical literal or a product of two physical literals (i.e. 10 um * 40 um). In VHDL, a physical literal is simply a number followed by a unit (10 mW, for

example). Each constraint expression restricts the legal value of the constraint type to a given range, for instance *power* $< 1\ V$.

VSPEC currently recognizes five constraint types: area, heat dissipation, power consumption, clock frequency and pin to pin timing. In a constraint expression, the first four of these constraint types are referenced with an identifier. Respectively, these identifiers are `area`, `heat`, `power` and `clock_frequency`. A slightly different notation is used to specify the final constraint type, pin to pin timing. The syntax for this type of constraint is:

⟨*timing_expression*⟩ ::= ⟨*input_pin*⟩ **<->** ⟨*output_pin*⟩

where input pin and output pin are identifiers that represent an input and an output port of the entity. Thus, an expression such as *input* <−> *output* $< 100\ us$ states that a change in the data at the input port is propogated to the output port in less than 100 microseconds.

As mentioned above, constraint values are either a physical literal or the product of two physical literals. Area is the only constraint type where a constraint value is the product of two physical literals. Area must be specified in this fashion with the two values representing the bounding box that the entity must fit into. All other constraint types have values that are physical literals.

There are several predefined units that are used for constraint values in VSPEC. The base units of these predefined units are meters for area, volts for power consumption, hertz for clock frequency and seconds for pin to pin timing. In addition to these base units, each of these units can also be expressed using the standard metric prefixes (i.e. area could be fm, um, mm, cm, m or km). VHDLalso allows the declaration of virtually any other physical type (see physical type definition in a VHDL reference [10, 16]).

In addition to the five pre-defined constraints, VSPEC users can create their own constraint types. At the present time, this has not been implemented in the VSPEC system, but this functionality is a part of the overall plan for the language.

3.5 Modifies Clause

⟨*modifies_clause*⟩ ::= MODIFIES ⟨*identifier_list*⟩ ;

The **modifies** clause is used to help build a list of signals and variables the entity will modify. The entity is guaranteed to change only the signals in this modifies list. The value of all other signals in the entity will be left unchanged. Since out mode port signals and all variables in the **state** clause would serve no purpose if the entity did not change them, all out mode port signals and variables in the **state** clause are automatically included in the modifies list. You may explicitly write them in the identifier list in the modifies clause if you desire, but this is an unneccessary step. On the other hand, global variables [2] and buffer/inout mode port signals may only be modified if they are included in the modifies list. It is an error to place in mode port signals in the modifies list since the definition of VHDL does not allow an entity to change the value of an input signal. Here is a simple example to clarify the signals and variables that will and will not occur in the modifies list:

```
ENTITY modifies_example IS
   PORT ( A : IN integer;
          B : OUT real;
          C, D : BUFFER bit;
          E, F : INOUT bit );
   STATE G : integer;
   MODIFIES C, E;
END modifies_example;
```

The list of signals/variables this entity will modify is C, E, B and G. C and E are included in this list because they are explicitely stated in the **modifies** clause. B is included because it is an output signal. All architectures of an entity must assign a value to all entity output signals. Thus, B is automatically included in the modifies list. G is included in this list for a similar reason. The

[2] Global variables were added to the 1993 version of VHDL. Previous definitions of the language did not contain global variables.

definition of VSPEC forces the entity to assign a value to all state variables so all state variables are automatically included in the modifies list.

3.6 Includes Clause

⟨*includes_clause*⟩ ::= INCLUDES ⟨*string_literal_list*⟩ ;

The **includes** clause is used to include a REFINE program in a VSPEC specification. This REFINE program defines the functions and types used in the specification and it helps map the VSPEC specification to its formal representation in the REFINE object base. A VSPEC specification may contain as many includes clauses as the user needs to describe the system. We have already seen an example of the **includes** clause in the search entity described in Section 3.1:

```
ENTITY search IS
  PORT (input : IN ARRAY OF INTEGER;
        key : IN INTEGER;
        output : OUT ARRAY OF INTEGER)
  REQUIRES sorted(input);
  INCLUDES "sort.re", "set.re";
END search;
```

In this example, the file "sort.re" contains the following REFINE definition of the **sorted** function:

```
function sorted ( input-seq : seq(integer) ) : boolean =
  if (size (input-seq) = 1) then
    true
  else
    ( input-seq(1) < input-seq(2) ) and sorted (rest(input-seq))
```

This is a boolean function that returns true when the input sequence is in order from smallest to largest. In formal logic, a boolean function is called a predicate. VSPEC users can define arbitrarily many predicates that are used to describe the observable behaviors of the system being designed.

Each of these predicates can appear in the `requires` or `ensures` clauses to describe a functional requirement of the system. All of the predicates that appear in these clauses must be defined in a REFINE file that is listed in one of the `includes` clauses in the entity where it is used.

3.7 Based On Clause

⟨*based_on_clause*⟩ ::= ⟨*vspec_type*⟩ BASED ON ⟨*refine_sort*⟩

The `based on` clause is used to map a data type used in VSPEC to its definition in REFINE. This definition in REFINE is called a sort. In the syntax above, vspec type is an identifier that refers to the data type used in VSPEC and refine sort is an identifier that represents the corresponding sort in REFINE.

The VSPEC system provides a built in mapping to REFINE for all predefined types in VHDL. This is accomplished by automatically including `based on` clauses for these predefined types in all VSPEC entities. The VHDL types `integer`, `real`, `boolean`, `character` and `string` map to their corresponding types in REFINE. The VHDL types `severity_level`, `bit` and `bit_vector` map to the following definitions in REFINE:

```
type severity_level = {'note, 'warning, 'error, 'failure}
type bit = {0, 1}
type bit_vector = seq(bit)
```

This means that the VSPEC systems adds `based on` clauses such as `integer BASED ON integer`, `character BASED ON char` and `bit_vector BASED ON bit_vector` to all VSPEC entities. In addition, VSPEC automatically includes a REFINE file that contains the three types above. With these clauses included in all VSPEC entities, the predefined types in VHDL may be used in any VSPEC specification.

4 Formal Representation of VSPEC

All VSPEC definitions can be transformed into a formal definition. This formal definition is based on an extension of domain theories defined in the CYPRESS [19] and KIDS [21, 20] systems. CYPRESS and KIDS are software synthesis systems that can be used to synthesize an efficient executable program from an algebraic specification. A domain theory is used to describe the problem to be synthesized. It consists of a tuple of the domain (D), range (R), input pre-condition ($I(x : D)$) and output post-condition ($O(x : D, z : R)$) commonly referred to as a *DRIO* model. In VSPEC, the *DRIO* model can be constructed using the following rules:

$D = d_1 \times d_2 \times \ldots \times d_n$ where each d_k is the sort (defined by the based on clause) representing the type associated with an **in**, **inout**, or **buffer** port or a state variable

$R = r_1 \times r_2 \times \ldots \times r_m$ where each r_j is the sort representing the type associated with an element in the modifies list (see Section 3.5)

$I(x : D) = I_v(x : D)$ where $I_v(x : D)$ is the logical sentence defined by the **requires** clause

$O(x : D, z : R) = O_v(x : D, z : R)$ where $O_v(x : D, z : R)$ is the logical sentence defined by the **ensures** clause

VSPEC is somewhat different from the specification languages that are normally used with CYPRESS and KIDS. A specification language for digital systems must provide a means for describing the performance constraints of the system. One way to do this would be to include these types of constraints in the output post-condition for the system. However, this is not the approach taken with VSPEC. Performance constraints have nothing to do with the function of the system so we feel it is appropriate to separate them from the functional requirements defined in the post-condition of the system (i.e. the **ensures** clause).

This is one reason the `constrained by` clause is included in VSPEC. The system's performance constraints are specified in the `constrained by` clause while the `ensures` clause describes the functional requirements of the system. The performance constraints can be represented in the formal model of VSPEC by extending the *DRIO* to a *DRIOC* model:

$C(c_1 : C_1, \ldots, c_n : C_n) = C_v(c_1 : C_1, \ldots, c_n : C_n)$ where c_k is a constraint variable such as heat or area, C_k is a sort associated with a constraint variable and C_v is the logical expression defined in the `constrained by` clause

The definitions in the *DRIOC*describe the system as a transformation mapping the current state and inputs into the next state and outputs such that when the input pre-condition is satisfied the output post-condition and constraints are also satisfied. Formally, this can written as:

$$\forall x : D \bullet I(x) \Rightarrow O(x, f(x)) \wedge C(c_1, \ldots, c_n) \tag{1}$$

where $f(x)$ is the transformation performed by the system. This axiom shows the relationship between the design, $f(x)$, and its requirements. In VSPEC, $I(x)$ is derived from the `requires` clause, $O(x, z)$ from the `ensures` clause and $C(c_1, \ldots, c_n)$ from the `constrained by` clause. In VSPEC $f(x)$ will be defined using behavioral VHDL. Finding $f(x)$ given I, O and C is the synthesis problem addressed by COMET. Proving the equation above is true for a given $f(x)$, I, O and C verifies that $f(x)$ is an implementation of the VSPEC specification.

5 Extended Example: 16-bit Move Machine

5.1 Problem Description

The Move Machine is a simple microprocessor whose instructions move data between CPU registers and main memory [18]. The computational units of the machine are assumed to be memory mapped. With this assumption, arithmetic and logical computations are performed as side effects of moving data to and from designated memory locations.

5.1.1 Physical Configuration

The physical storage components of the Move Machine are a main memory array and a set of registers. The registers consist of an instruction pointer, an instruction register, and an array of general purpose registers.

In this example, a 16-bit Move Machine is specified. The configuration used has 16 general purpose registers, each 16 bits long. The main memory size is 512 bytes (256 16-bit values), requiring 8-bit addressing. The instruction pointer is 8 bits and the instruction register is 16 bits.

5.1.2 Instruction Format

The instructions of the 16-bit Move Machine have four fields:

- A two bit op-code. The four operations that the Move Machine has are: load, store, jump, and halt.
- A two bit addressing mode which determines how the effective address is specified in the instruction. The four addressing modes are: absolute, immediate, indirect, relative.
- A four bit register identification to specify which register is to take part in the operation.

- An eight bit effective address which, in conjunction with the addressing mode, determines which memory location takes part in the instruction.

5.1.3 Processor Operation

The I/O interface to the Move Machine consists of a start signal, a finished signal and a clear signal. When the start signal is received, the processing cycle begins. When the machine halts (executes a halt instruction), the finished signal is set. The clear signal resets the machine and prepares it to receive the start signal.

The Move Machine has a three phase processing cycle. In the first phase, the instruction referenced by the instruction pointer is fetched from memory. In the second phase, the effective address is calculated according to the specified addressing mode and the instruction pointer is incremented to reference the next instruction. In the third phase, the fetched instruction is executed.

5.2 Specification of the Move Machine

The first step in specifying the behavior of the Move Machine is to define abstract data types in REFINE. These types and there associated operations will provide the vocabulary necessary to describe the behavior of the Move Machine. Once this foundation is laid, defining the VSPEC interface specification can begin. First, the input, output, and state variables are specified. Then the desired behavior is described using the appropriate VSPEC clauses.

5.2.1 Abstract Types and Operations

Abstract data types and operations are specified using the REFINE language. REFINE supports a host of set theoretic data types, such as sets, sequences, tuples, and maps. Sets and sequences represent unordered and ordered collections of objects, respectively. Tuples are an ordered collection

of related data, similar to a VHDL record. Maps represent a functional relation between two types. Formally, they are a set of 2-tuples such that $M(x) = y$ means that $(x, y) \in M$. Some additional REFINE constructs will be introduced as they are used in the example. REFINE has a complete array of operations for the predefined data types. For a more complete explanation of REFINE types and operations, see the REFINE User's Manual [17].

Figure 3 shows the REFINE specification of the Move Machine data types and operations. The first section in Figure 3 shows the predefined VHDL types available for use within the REFINE specification. These are shown for reference, to make the example self-contained. The predefined VHDL types are shown in all caps whenever they are used. The next section in Figure 3 is a group of constant declarations that define the hardware configuration of the Move Machine.

The next group of declarations are the abstract data types. First, the `Word` type is introduced as a set of `BIT_VECTOR`. Next, the `Address` type is defined as an integer subrange. Variables of type `Address` will have an integer value between `0` and `MM_Size-1`. The type `Memory_Array` is defined as a map from `Addresses` to `Words`. This means that for a `Memory_Array`, `M`, and an `Address`, `x`, the `Word` at memory location `x` is simply `M (x)`. Notice that the size of a `Memory_Array` is restricted by the upper bound of the `Address` integer range. `Register_Array` and `Register_Id` are specified in the same manner as `Memory_Array` and `Address`.

The abstract type `Operation` is defined to describe the four possible Move Machine operations. This is done using a symbol. Symbols are a REFINE type used to represent an abstract value. They are not strings or sequences of characters. Each symbol literal is a unique atomic value. The Move Machine's four addressing modes are similarly represented by the `Add_Mode` type.

The `Instruction` type is a 4-tuple representing the four fields of the instruction. The tuple values are accessed in the same manner as fields of a record. The `op_code` value for an `Instruction`, `i`, is simply `i.op_code`.

The last data type specified is `Proc_State`. This type is used to represent the abstract states of the

```
%-- REFINE move_mc_types.re -- abstract type for the Move Machine.
% The following lines are needed in all Refine programs
!! in-package("RU")
!! in-grammar('user)

%  predefined VHDL types and operations
%  type BIT = boolean
%  type BIT_VECTOR = seq(BIT)
%  function bits_to_int(b:BIT_VECTOR) : INTEGER

% Move Machine constant declarations
  constant MM_Size : INTEGER = 256
  constant Register_Array_Size : INTEGER = 16
  constant Word_Size: INTEGER = 16

% Move Machine type declarations

  type Word = BIT_VECTOR
  type Address = {0..MM_Size-1}                  % integer range
  type Memory_Array = map(Address,Word)
  type Register_Id = {0..Register_Array_Size-1}  % integer range
  type Register_Array = map(Register_Id,Word)
  type Operation = SYMBOL
  type Add_Mode = SYMBOL
  type Instruction =
          tuple(op_code : Operation, addr_mode : Add_Mode,
                reg_id : Register_Id, eff_addr : Address)

  type Proc_State = SYMBOL

% Operations over the Move Machine types
  function Word_to_Instr(data : Word) : Instruction =
        < Decode_Op(subseq(data,0,1)),
          Decode_AM(subseq(data,2,3)),
          bits_to_int(subseq(data,4,7)),
          bits_to_int(subseq(data,8,15)) >

  function Decode_Op(data : seq(BIT)) : Operation
        computed-using  data = [false,false] => Decode_Op(data) = 'load,
                        data = [false,true] => Decode_Op(data) = 'store,
                        data = [true,false] => Decode_Op(data) = 'jump,
                        data = [true,true] => Decode_Op(data) = 'halt

  function Decode_AM(data : seq(BIT)) : Add_Mode
        computed-using  data = [false,false] => Decode_AM(data) = 'absolute,
                        data = [false,true] => Decode_AM(data) = 'immediate,
                        data = [true,false] => Decode_AM(data) = 'indirect,
                        data = [true,true] => Decode_AM(data) = 'relative
```

Figure 3: Move Machine data types and operations.

Move Machine's operation. The three processing phases, fetch, decode, and execute, are represented along with start and stop states. The allowable actions of the Move Machine's behavior will be expressed as transitions between these five processor states.

The last section in Figure 3 is the specification of `Word_to_Instr`, an operation that converts between `Words` and `Instructions`. This conversion will be necessary because instructions are stored in memory as words. Notice that syntax of REFINE permits simply equating the function with a tuple construct. The values of each of the tuple fields are themselves function calls. The REFINE `subseq` operation is used to extract a smaller sequence from an existing sequence. This operation can be used with the type `Word`, because it is a `BIT_VECTOR` which is a sequence of `BITS`. The functions `Decode_Op` and `Decode_AM` are used to precisely define the operation and addressing mode deciding scheme.

5.2.2 VSPEC Interface Specification

This section contains a detailed description of the interface specification for the Move Machine. The entire specification is shown in Figure 4. We will describe each section of this specification separately, starting with the port declaration. This is where the entity `move_mc` is created and its I/O ports are declared in standard VHDL syntax. The start and clear signals are defined as inputs and the finished signal is defined as an output. The Move Machine port declaration is:

```
entity move_mc is
  port (Start: in BIT;        -- Begin processing
        Clear: in BIT;        -- Restart processing
        Finished: out BIT);   -- Processing completed
```

The VSPEC `includes` clause follows the port declaration:

```
includes "move_mc_types.re";
```

```
entity move_mc is
  port (Start: in BIT;        -- Begin processing
        Clear: in BIT;        -- Restart processing
        Finished: out BIT);   -- Processing completed

includes "move_mc_types.re";

state
    phase: Proc_State,        -- Abstract Processor State
    Memory : Memory_Array,    -- Main Memory
    IP : Address,             -- Instruction Pointer
    IR : Instruction,         -- Instruction Register
    RGST : Register_Array,    -- General Purpose Registers
    EA : Address,             -- Effective Address

ensures
    phase = start implies (Start = '1' implies phase'post = fetch)
        and (Start = '0' implies phase'post = start)
        and IP'post = 0
        and Memory'post = Memory and RGST'post = RGST
    and
    phase = fetch implies IR'post = Word_to_Instr(Memory(IP))
        and phase'post = decode and Memory'post = Memory
        and RGST'post = RGST and IP'post = IP
    and
    phase = decode implies phase'post = execute
        and (IR.addr_mode = absolute implies
             EA'post = IR.eff_addr and IP'post = IP + 1)
        and (IR.addr_mode = immediate implies
             EA'post = IP + 1 and IP'post = IP + 2)
        and (IR.addr_mode = indirect implies
             EA'post = Word_to_Instr(Memory(IR.eff_addr)).eff_addr
             and IP'post = IP + 1)
        and (IR.addr_mode = relative implies
             EA'post = IP + IR.eff_addr and IP'post = IP + 1)
        and Memory'post = Memory and RGST'post = RGST and IR'post = IR
    and
    phase = execute implies
        (IR.operation = load implies RGST(IR.reg_id)'post = Memory(EA)
            and forall(x:Register_Id)
                    (x /= IR.reg_id implies RGST(x)'post = RGST(x))
        and (IR.operation /= load implies RGST'post = RGST)
        and (IR.operation = store implies Memory(EA)'post = RGST(IR.reg_id))
             and forall(x:Address)(x /= EA implies Memory(x)'post = Memory(x))
        and (IR.operation /= store implies Memory'post = Memory)
        and (IR.operation = jump implies IP'post = EA)
        and (IR.operation /= jump implies IP'post = IP)
        and (IR.operation = halt implies phase'post = stop)
        and (IR.operation /= halt implies phase'post = fetch))
    and
    phase = stop implies Finished'post = '1'
        and (Clear = '0' implies phase'post = stop)
        and (Clear = '1' implies phase'post = start)
        and Memory'post = Memory and RGST'post = RGST
    and
    phase /= stop implies Finished'post = '0';
end move_mc;
```

Figure 4: VSPEC interface specification for the Move Machine

The `includes` clause states that this specification will use abstract types and operations defined in the file `move_mc_types.re`, which was described in the previous section.

The behavior of the Move Machine is specified by describing the allowable transactions between processor states [14]. To do this, we must first definine the information that determines the processor state. The Move Machine has a three phase processing cycle which can be viewed as processor states. The addition of a start and a stop state defines a set of states which uniquely describes the status of the Move Machine at any moment in time. The abstract type `Proc_State` was defined specifically for this purpose. Therefore, the `state` clause contains the variable `phase` of type `Proc_State` to model the processor state:

```
state
    phase: Proc_State,          -- Abstract Processor State
    Memory : Memory_Array,      -- Main Memory
    IP : Address,               -- Instruction Pointer
    IR : Instruction,           -- Instruction Register
    RGST : Register_Array,      -- General Purpose Registers
    EA : Address,               -- Effective Address
```

Naturally, the values of the registers and main memory are of interest when observing the behavior of the processor. Variables of these type are declared in the `state` clause to model these physical structures. In addition, any internal signals that are used to communicate between processor states must be declared as state variables. The effective address is calculated in the decode phase but it is used in the execute phase. Therefore, the variable `EA` of type `Address` is declared to store the effective address between states.

Given a set of input and state variables, the VSPEC `ensures` clause can be used to specifiy the allowable changes to the output and state variables. In this way, the behavior of the Move Machine is defined. The Move Machine `ensures` clause is structured according to the value of the `phase` variable. This clarifies the specification of the state transactions that are allowed during each phase

of processor execution. The allowable transactions for each phase are then conjuncted together to provide a complete behavioral specification.

The permissible next state values must be explicitly constrained for each state variable. If a state variable is not constrained, then presumably it is allowed to take on any value of the associated type. It is not assumed that unconstrained variables remain unchanged. Constraining a variable's behavior is accomplished using the VSPEC **implies** operator to define the next state values that are possible during each processor phase. In this example, the next state values are determinant, but this is not a necessary condition. Non-determinism can be modeled by disjuncting allowable next state values.

The first part of the **ensures** clause specifies what transactions are allowed during the start phase. While in the start phase, the processor is simply waiting for the start signal to begin processing. If the processor does not receive the start signal, it stays in the start phase. This constraint on the next state value of the phase variable (`phase'post`) is specified by the first two conjuncts implied by the start phase. Note that the notation <*variable*>`'post`, where <*variable*> is the identfier for the variable, is used to refer to the value of the variable after the transaction occurs. Here is the part of the **ensures** clause which describes the start phase:

```
phase = start implies (Start = '1' implies phase'post = fetch)
    and (Start = '0' implies phase'post = start)
    and IP'post = 0
    and Memory'post = Memory and RGST'post = RGST
```

The conjunct, `IP'post = 0`, states that the first instruction will be retrieved from memory location 0. The final two conjuncts specify that the main memory and register values must remain unchanged during this processing phase. Without these constraints, the specification would be satisfied by an implementation where the memory and registers values arbitrarily change during this state, which is not the desired behavior. Notice that the state variable EA is not constrained during this phase.

At this point, the EA variable does not contain any information which will effect the future state of the machine. Therefore, the specification need not be constrained to retain the value of this variable.

The Move Machine behavior during the fetch phase is described by:

```
phase = fetch implies IR'post = Word_to_Instr(Memory(IP))
    and phase'post = decode and Memory'post = Memory
    and RGST'post = RGST and IP'post = IP
```

During the fetch phase, the instruction pointer is updated to ontain the interpretation of the word at memory location IP. Here, the interpretation is performed by the Word_to_Instruction function defined in the previous section. The next processing phase is specified to be decode, while the memory and remaining register values remain unchanged.

The state changes which occur during the decode phase hinge on the addressing mode. Therefore, the majority of the specification of the decode phase is structured around the value of IR.addr_mode:

```
phase = decode implies phase'post = execute
    and (IR.addr_mode = absolute implies
         EA'post = IR.eff_addr and IP'post = IP + 1)
    and (IR.addr_mode = immediate implies
         EA'post = IP + 1 and IP'post = IP + 2)
    and (IR.addr_mode = indirect implies
         EA'post = Word_to_Instr(Memory(IR.eff_addr)).eff_addr
         and IP'post = IP + 1)
    and (IR.addr_mode = relative implies
         EA'post = IP + IR.eff_addr and IP'post = IP + 1)
    and Memory'post = Memory and RGST'post = RGST and IR'post = IR
```

The effective address, EA and instruction pointer, IP, are updated according to the current addressing mode. The next phase is specified to be the execute phase. The main memory, the CPU registers and the instruction register are unchanged.

The Move Machine behavior during the execution phase depends upon the fetched operation. This part of the specification is determined by the Move Machine operations:

```
phase = execute implies
    (IR.operation = load implies RGST(IR.reg_id)'post = Memory(EA)
        and forall(x:Register_Id)
                (x /= IR.reg_id implies RGST(x)'post = RGST(x))
    and (IR.operation /= load implies RGST'post = RGST)
    and (IR.operation = store implies Memory(EA)'post = RGST(IR.reg_id))
         and forall(x:Address)(x /= EA implies Memory(x)'post = Memory(x))
    and (IR.operation /= store implies Memory'post = Memory)
    and (IR.operation = jump implies IP'post = EA)
    and (IR.operation /= jump implies IP'post = IP)
    and (IR.operation = halt implies phase'post = stop)
    and (IR.operation /= halt implies phase'post = fetch))
```

For a load operation, the register identified by the current instruction is assigned the value of the memory location referenced by the effective address. This is easily specified by: `RGST(IR.reg_id)'post = Memory(EA)`. However, it is also necessary to specify that the remaining registers do not change. This is the purpose of the second conjunct implied by the load operation. Using the VSPEC `forall` construct, it states that every register that is not involved in the load operation retains its value. When the instruction does not specify a load operation, the values of the register array do not change.

Similarly, for a store operation, the specification states that the specified memory location changes while the rest remain unchanged. The jump operation only effects the value of the instruction pointer. A halt operation causes the next phase to be the stop phase. Any other operation results in the processing returning to the fetch phase.

During the stop phase, the processor sets the finished signal and monitors the clear signal. The stop phase is specified by:

```
phase = stop implies Finished'post = '1'
    and (Clear = '0' implies phase'post = stop)
    and (Clear = '1' implies phase'post = start)
    and Memory'post = Memory and RGST'post = RGST
and
phase /= stop implies Finished'post = '0';
```

The next phase is determined by the clear signal. This part of the specification also constrains the finished signal to be low during every other phase.

The full behavior of the Move Machine is modeled by conjuncting the specifications for the individual phases. Figure 4 shows the entire specification for the Move Machine.

6 Related Work

VSPEC uses an axiomatic specification technique based on the approach developed for the Larch [7] family of specification langauges. On the surface, VSPEC is a prototype Larch interface language for VHDL. Thus, many of its constructs can also be found in other Larch interface languages, most specifically LM3 [12], an interface language for Modula-3. Currently, VSPEC is not a true interface language as its semantics are defined using REFINE rather than the Larch Shared Language (LSL). However, the general concept of a language specific axiomatic interface language in combination with a means for writing auxiliary specification is prominent.

Odyssey Research Associates (ORA) is developing a Larch interface language for VHDL [11]. This language differs from VSPEC because it is targeted for formal analysis of the system rather than for synthesis. ORA is attempting to generate a formal semantics for VHDL using LSL for proving correctness. This approach is adopted from the Ada work previously done in the Penelope project [4]. In ORA's interface language, time is the only non-functional constraint directly represented. Rather than placing constraints on pin-to-pin timing, an absolute time based termporal logic is used to specify the an entity's function. One can specify that a predicate $P(x)$ must be true

at time t using the notation "$P(x)@t$". Thus, a system's timing constraints are intermingled in the definition of the function of the system. The VSPEC notation specifies time intervals as constraints independent of system function. In principle, separation of concerns is a design goal for any specification language. In practice, including temporal aspects in the functional specification requires use of theorem provers capable of temporal reasoning. Currently, there are few such production quality provers. In VSPEC, information needed for constraint verification is included, but one may choose characteristics for verification.

VAL [1] is another attempt to annotate VHDL. VAL (VHDL Annotation Language is based on similar work done with Anna for Ada programs [15]. VAL differs from VSPEC because it is an annotation of a specific VHDL design rather than a representation of the requirements for a system not yet designed. VSPEC clauses may access only `ports` defined by the `entity` and variables defined locally in the specification. VAL annotations exist throughout the VHDL specification and formally document its behavior. Any local variable may be referenced in a VAL annotation. Specific aspects of both the structural and behavioral implementation are documented in the VAL annotation. VAL's intent is to document a design for verification where VSPEC's intent is to define requirements for a system.

7 Current Status and Future Directions

Current VSPEC research involves pursuing domain specific support for prototype synthesis. The role of VSPEC in the COMET system is as a requirements specification language and as input to synthesis tools. Thus, we are working to develop techniques to transform VSPEC into behavioral and structural VHDL. An important related technology transfer issue is developing a handbook of reusable specifications. In the Larch tradition, a handbook is simply a collection of reusable theories defined in the shared language. Handbook theories represent commonly used structures, algorithms and characteristics as well as domain specific information. For VHDL theories representing standard

VHDL types, low level logic functions, signal attributes and conversion routines are some libraries currently being implemented. Theories for pin-to-pin timing, heat dissipation, power consumption, area and clock speed have been implemented to support constraint checking during the design process.

We are beginning an effort to make VSPEC a true Larch interface language. Specifically, defining each of its constructs using LSL and developing tools for manipulating the specifications. Of particular interest is the representation of parallel components. Each `entity` structure exists asynchronously in parallel with other `entities` in the same design. representing such parallelism in VSPEC is a current area of research.

A prototype VSPEC parser has been developed and will be used to drive synthesis tools and the translation from VSPEC to LSL. The parser is developed using the SOFTWARE REFINERY's DIALECT tool and parses VHDL93 with VSPEC extensions into an abstract syntax tree. This data structure serves as the basis for interfacing VSPEC with other tools.

8 Acknowledgments

Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615-93-C-1316. The authors wish to thank Wright Labs and ARPA for their continuing support and direction.

References

[1] L. Augustin, D. Luckham, B. Gennart, Y. Huh, and A. Stanculescu. *Hardware Design and Simulation in VAL/VHDL.* Kluwer Academic Publishers, Boston, MA, 1991.

[2] Yoonsik Cheon and Gary T. Leavens. A Quick Overview of Larch/C++. *Journal of Object-Oriented Programming*, 7(6):39–49, October 1994.

[3] Stephen J. Garland, John V. Guttag, and James J. Horning. Debugging Larch Shared Language Specifications. Technical Report 60, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, July 1990.

[4] David Guaspari. Penelope, an Ada Verification System. In *Proceedings of Tri-Ada '89*, pages 216–224, Pittsburgh, PA, October 1989.

[5] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal Verification of Ada Programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.

[6] John V. Guttag and James J. Horning. Introduction to LCL, A Larch/C Interface Language. Technical Report 74, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, July 1991.

[7] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, NY, 1993.

[8] John V. Guttag, James J. Horning, and Andres Modet. Report on the Larch Shared Language: Version 2.3. Technical Report 58, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, April 19990.

[9] C.A.R. Hoare. Algebra and Models. *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 18(5):1–8, December 1993.

[10] Institute of Electrical and Electronics Engineers, Inc., 345 East 47th St., New York, NY 10017. *VHDL Language Reference Manual*, 1994.

[11] D. Jamsek and M. Bickford. Formal Verification of VHDL Models. Technical Report RL-TR-94-3, Rome Laboratory, Griffiss Air Force Base, NY, March 1994.

[12] Kevin D. Jones. LM3: A Larch Interface Language for Modula-3, A Definition and Introduction Version 1.0. Technical Report 72, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, June 1991.

[13] Knowledge Based Software Engineering Laboratory, University of Cincinnati. *VSPEC Language Reference Manual*, 1994. In Preparation.

[14] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[15] D. Luckham and F. von Henke. An Overview of Anna, a Specification Language for Ada. *IEEE Software*, 2(2):9–22, March 1985.

[16] Douglas L. Perry. *VHDL.* McGraw-Hill, Inc., New York, NY, 1991.

[17] Reasoning Systems Inc., Palo Alto, CA. *Refine User's Guide, Version 3.0*, May 1990.

[18] Jayanta Roy, Nand Kumar, Rajiv Dutta, and Ranga Vemuri. DSS: A Distributed High-Level Synthesis System. *IEEE Design&Test of Computers*, pages 18–32, June 1992.

[19] D. Smith. Top-down Synthesis of Divide-and-Conquer Algorithms. *Artificial Intelligence*, 27(1):43–96, Sept. 1985.

[20] D. Smith. Algorithm Theories and Design Tactics. *Science of Computer Programming*, 14:305–321, 1990.

[21] D. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, Sept. 1990.

[22] R. Vemuri, H. Carter, and P. Alexander. Board and MCM Level Synthesis for Embedded Systems: The COMET Cosynthesis Environment. In *Proceedings of the First Annual RASSP Conference*, pages 124–133, Arlington, VA, August 15-18 1994.

APPENDIX C:

Pipelined Scheduling of Hardware-Software Codesigns *

Karam S. Chatha and Ranga Vemuri
Department of ECECS
University of Cincinnati
Cincinnati, Ohio 45221-0030
Email: ranga.vemuri@uc.edu

Abstract

This paper discusses a scheduling technique for pipelined hardware-software codesigns. The technique uses scheduling and retiming to optimize the performance of a given codesign. The paper presents heuristics for scheduling and retiming which aim to optimize the throughput and memory requirements of a given codesign. The effectiveness of the technique is demonstrated by experimentation.

1 Introduction

Hardware-Software codesigns are characterized by strict performance constraints. The codesign process partitions the system specification in to interacting hardware (HW) and software (SW) tasks which exhibit the desired behavior and satisfy the performance requirements. In a typical codesign flow the HW-SW partitioner and the scheduler execute in an iterative fashion till a constraint satisfying design is obtained. Many digital signal processing (DSP) algorithms are loop oriented, which makes them suitable for pipelined codesign implementation. In this paper we present a technique which optimizes the throughput and memory requirements of pipelined codesigns by scheduling and retiming.

The system specification is captured in an intermediate graph format called the *Data Dependency Graph* (*DDG*). The vertices of the graph represent the tasks and the edges represent the data dependencies among the various tasks. The *granularity* of the tasks is determined by the user. The execution times of the tasks on the SW processor and in HW are obtained by *profiling* and HW performance estimation respectively [5]; and are stored in the graph representation. The edges contain information about the number of variables across a dependence. The DDG representation will be discussed in detail in Section 3.

The codesign architecture consists of a single general purpose SW processor, a single application specific integrated chip (ASIC) and a shared memory (Figure 1). The SW processor and ASIC are connected to the shared memory through the system bus. The general purpose processor and the ASIC themselves are non-pipelined with respect to task execution, that is a new task cannot begin execution before the previous one has finished. Communication between tasks bound to different resources (that is from SW to HW or HW to SW) takes place through the shared memory. Also data transfers between two tasks bound to ASIC takes place through the shared memory. The shared memory is *exclusive read exclusive write* and therefore no two tasks can either read or write at the same time.

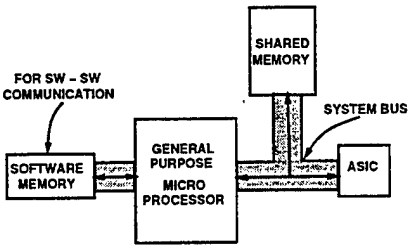


Figure 1: Codesign Architecture

The throughput of loop-oriented codesigns can be maximized by obtaining a pipelined implementation. The drawback of pipelining is that it increases the memory requirement of the design. Consider the DDG shown in Figure 2. It consists of three tasks shown as bubbles in the figure. The binding and execution

*This work was partially supported by the ARPA RASSP program and monitored by the Wright Lab, US-AF under contract number F33615-93-C-1316 and ARPA HPCC program monitored by the FBI under contract number J-FBI-93-116

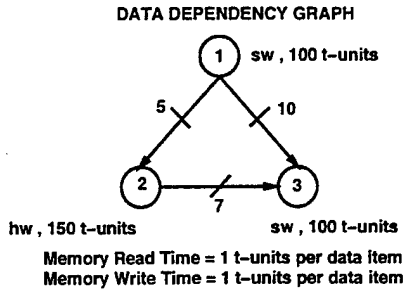


Figure 2: DDG Example

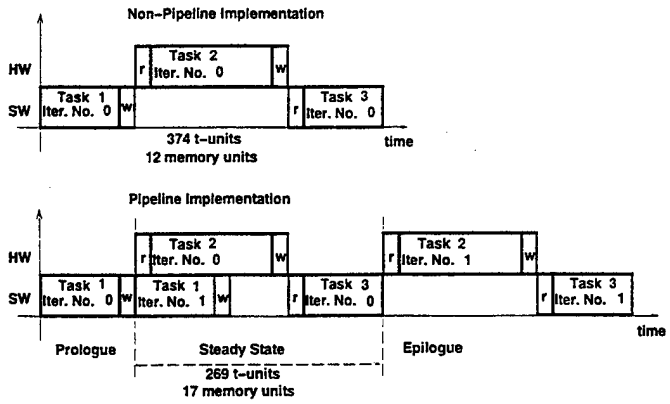


Figure 3: Non-pipeline and Pipeline Implementation

times of the tasks are shown beside each bubble. The data dependencies are shown as directed edges and the data items transferred by each dependency are written next to the edges. The memory read and write times are also shown in the figure. We assume that the DDG is executed a number of times inside a loop. The non-pipeline and pipeline implementations of the design are shown in Figure 3. The rectangles in Figure 3 represent the execution of various tasks. Each rectangle contains the task number and iteration number of the loop to which it belongs. The small rectangles with "r" and "w" represent memory read and write respectively. We assume that a task while executing needs memory space for both its read set and write set. The read (write) set of a task is the set of data items read (written) by the task. As can be seen from the figure the non-pipeline implementation takes 374 t-units to complete one iteration of the loop and it requires 12 memory units. The pipeline implementation overlaps the execution of tasks belonging to different iterations of the loop. When fully loaded the steady state completes one iteration in 269 t-units, a definite improvement on the previous design. But it requires 17 memory units for its execution.

The paper presents a technique for optimizing the performance of pipelined codesign. The technique uses a *list based scheduler* [1] and *retiming transformations* [2] to obtain a pipelined codesign. The paper presents heuristics for both scheduling and retiming which try to maximize the throughput of the design while trying to minimize the memory requirements.

The paper is organized as follows. In Section 2 we discuss previous work, in Section 3 we describe the DDG representation, Section 4 presents the pipeline scheduling technique, the experimental results are in Section 5 and finally Section 6 concludes the paper.

2 Previous Work

Based on their application area existing codesign methodologies can be broadly classified in to two categories. Category one would include methodologies oriented towards real time reactive systems [7][8][11][13]. Scheduling in reactive systems is done to ensure that time constraints and data dependencies between different processes are satisfied [12]. Category two would contain methodologies that are meant for data processing applications [10]. Design methodologies for such applications use scheduling to maximize the throughput of a given codesign partition. Our codesign flow would fall into category two. In this paper we present a scheduling heuristic for optimizing throughput and memory requirements of a design. Pipelining is an effective way for maximizing the throughput of a loop oriented design. Other research [9] has used pipelining for mixed applications which include both control constructs and data processing tasks. We use retiming [2] to generate pipeline designs. The formalism for the problem description and the general technique is described in [3] and we use the same in our paper. Retiming heuristics in [3] aim at obtaining pipelined implementations with optimum throughput. In this paper we present a scheduler interacting with a retimer to optimize both throughput and memory requirements of pipelined codesign applications.

3 Data Dependency Graph

The input specification is captured by an intermediate graph called the *Data Dependency Graph* (DDG). It represents the tasks by vertices and the data dependencies between tasks by directed edges. The vertices have information about the task binding (HW or SW), HW execution time and SW execution time. The

edges have information about the number of variables in a dependence. Since we are interested in pipelining the design, we associate with each vertex an iteration index (λ) and with each edge a dependency distance (δ) [3]. The iteration index $\lambda(u)$, of a task u indicates that at the i^{th} iteration of the steady state, instance of task u belonging to the $(i + \lambda(u))$ iteration of the loop is executed. For example consider the pipelined design in Figure 3. In the first iteration of the steady state, instance of task 1 belonging to the second iteration of the loop is executed, hence $\lambda(task1) = 1$. The dependence distance of an edge e, $\delta(e)$ indicates the number of iterations of the steady state traversed by that edge. In the pipelined implementation in Figure 3, the data produced by task 1 at the i^{th} iteration of the steady state is consumed by task 2 at the $(i+1)^{th}$ iteration of the steady state. Hence the dependence distance of edge $(1, 2)$ is $\delta(1, 2) = 1$. We now formalize the DDG representation as follows:
A DDG is a 4-tuple $DDG = G(V, E, \lambda, \delta)$, where:

- *V is the set of vertices. Each vertex $u \in V$ represents a task. For each task $u \in V$ we have the following information available to us :*
  - *u_{bind} : The binding of the task, that is whether its going to be implemented in HW or SW.*
  - *u_{sw} : The SW runtime of the task for a particular input data on the general purpose processor.*
  - *u_{hw} : The HW runtime of the task if it were to be implemented as an ASIC for the same input data.*
- *E is the set of directed edges. Each $e = (u, v) \in E$ represents a data dependence between tasks u and v. Every edge has information about the number of variables (e_{var}) represented by the dependence.*
- *λ and δ are two mappings, $\lambda : V \rightarrow I\!N$ and $\delta : E \rightarrow I\!N$, representing the iteration index (λ) and the number of iterations traversed by the dependence (δ). $I\!N$ is the set of natural numbers.*

Initially, $\forall u \in V$, $\lambda(u) = 0$. Notice that the representation has no control flow constructs; it is strictly data flow. Now we explain and formalize terms and expressions that we will use in the rest of the paper.

The latency of a task u, L_u, is the total execution time of the task. It is the sum of the task's read time, execution time on the particular resource that its been bound to and write time. The read (write) time of a task is the product of the number of variables read (written) by the task and the memory read (write) time. Since we have only two resources, the execution time for a task on a resource is u_{sw} (if $u_{bind} = sw$) or u_{hw} (if $u_{bind} = hw$).

For a particular pipeline implementation, the *initiation interval II*, is the time taken for one iteration of the steady state. For example in Figure 3, the pipelined implementation has $II = 269$ t-units. Given a DDG and an architecture its possible to establish a lower bound on the initiation interval. This is called the *minimum initiation interval, MII.* The MII is limited by two factors. Firstly the architecture resources limit the MII. This is called the *resource constrained MII*, $ResMII$. For example the DDG in Figure 2 requires at least 212 t-units to execute tasks 1 and 3 which are bound to SW. The SW resource constrained MII, $ResMII_{SW}$ is given by the sum of latencies of all tasks bound to SW implementation. Similarly, HW resource constrained MII, $ResMII_{HW}$ is the sum of latencies of all tasks bound to HW implementation. $ResMII$ for the DDG is then the maximum of the two, that is $ResMII = max(ResMII_{SW}, ResMII_{HW})$ Secondly, recurrences or cycles in the DDG also limit MII. This is called the *recurrence constrained MII*, $RecMII$. For example consider the DDG example shown in Figure 2. Let us assume that we add an extra dependency $e = (2, 1)$ with $\delta(2, 1) = 1$ to the DDG. In such a case the pipelined implementation in Figure 3 becomes invalid. This is because the instance of task 1 belonging to the second iteration cannot start executing before the the instance of task 2 belonging to the first iteration of the loop. This constraint is introduced because of the recurrence present in the DDG. The $RecMII_r$ for a recurrence r, is given by the ratio of the sum of the latencies of the tasks in the recurrence to the sum of the weights (δ) of all the dependencies in a recurrence. A graph may have more than one cycle, and $RecMII$ is then the maximum of the $RecMII_r$ due to each one of them, that is $RecMII = max(RecMII_r)$, for all the recurrences r in the DDG. The MII is then the maximum of $ResMII$ and $RecMII$, that is $MII = max(ResMII, RecMII)$. The maximum execution throughput of a DDG, $MaxTh$ is the maximum iterations of the steady state possible in one time unit. Its given by:

$$MaxTh = \frac{1}{MII}$$

4 Pipeline Scheduling Technique

The objective of the technique is to obtain a pipeline schedule of the the DDG which has MII as

its initiation interval and which requires least amount of shared memory. The pipeline schedule of the DDG determines the steady state of the pipeline. The flow diagram of the technique is shown in Figure 4. The inputs to the pipeline scheduler are the partitioned DDG, the codesign architecture and a desired upper bound on initiation interval, $MaxII$. The pipeline scheduler first calculates the MII for the design. It then tries to schedule the DDG in MII time. If its unsuccessful it selects a dependency to be retimed. Retiming as we will see later transforms a schedule constraining dependency into a free scheduling dependency which does not constrain the scheduler. In this process however, it increases the iteration indices of some tasks. Hence retiming produces a DDG with tasks belonging to different iterations of the steady state. In other words retiming produces a pipelined DDG. This inner loop of scheduling and retiming continues till a successful schedule is found or all the dependencies have been retimed. In the latter case we increase the initiation interval and try scheduling again. We set the increment factor to the maximum of the following two values: one time unit or one percent of MII. We exit the outer loop when the initiation interval II becomes greater than the user specified $MaxII$.

The inputs to the scheduler are the DDG and the expected initiation interval II. The objective of the scheduler is to obtain a pipeline schedule of the DDG in II time using the least amount of shared memory. The schedule is an assignment of start times to tasks, $S(u)$, such that for all tasks u in the graph $0 \leq S(u) \leq II$ [3]. For a dependency $e = (u, v)$, the schedule time of u and v must honor the data dependence, ie $S(v) + \delta(u,v) \times II \geq S(u) + L_u \Rightarrow S(v) \geq S(u) + L_u - \delta(u,v) \times II$. Also there should be enough resources and shared memory to execute a task scheduled at a particular time instance. The memory requirement of a task during execution is the total memory required by the variables in the task's read set and write set. The pipeline schedule of a task is then formalized as below:

For a given II, a pipeline schedule of $DDG = G(V, E, \lambda, \delta)$ is an integer labeling, $S \rightarrow \mathbb{N}$ which fulfills the following conditions :

- $\forall u \in V, 0 \leq S(u) \leq II$.
- $\forall (u,v) \in E, S(v) \geq S(u) + L_u - II \cdot \delta(u,v)$, *that is all dependences must be honored.*
- *There are sufficient resources (HW and SW) to execute the task scheduled at a particular time instant.*

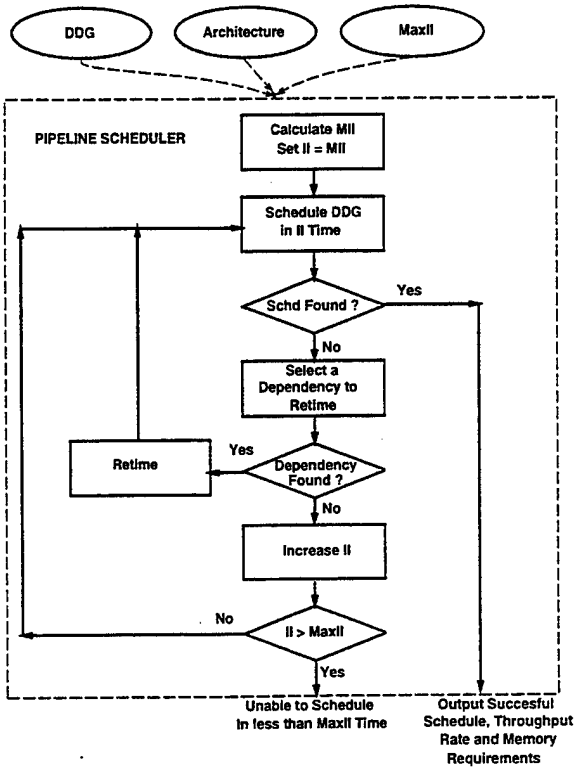


Figure 4: Pipeline Scheduling Technique

- *There is sufficient memory to execute the task scheduled at a particular time instant.*

Schedule Constraining Dependencies. For a given initiation interval II, the data dependencies in a DDG can be classified in to *positive scheduling dependencies (PSDs), negative scheduling dependencies (NSDs) or free scheduling dependencies (FSDs)* [3]. A dependency (u, v) is a PSD if $L_u - II \cdot \delta(u,v) > 0$. A dependency is a FSD or NSD if $L_u - II \cdot \delta(u,v) < 0$. $PSDs$ constrain scheduling since they make $S(v) > S(u)$, in other words task v must be scheduled later than task u. $FSDs$ do not constrain scheduling. $NSDs$ could constrain a schedule if pipelined resources are used or if an iteration of the steady state begins before the previous one finishes (non-rectangular schedule). Since neither of the two conditions are true in our case, $NSDs$ do not constrain the schedule. The set of schedule constraining dependencies E^S is then given by:

$$E^S = \{(u,v) \in E | L_u - II \cdot \delta(u,v) > 0\}$$

$PSDs$ are also called intra loop dependencies (or $ILDs$) and $FSDs$ and $NSDs$ are together called as loop carried dependencies (or $LCDs$). A dependency (u, v) is a ILD if $\delta(u, v) = 0$ and it is a LCD if $\delta(u, v) > 0$.

Given a set of schedule dependencies we can define two properties for every task. The first one called the height of the task, $H(u)$ gives the as soon as possible (ASAP) schedule time of a task. The second one called the depth of a task, $D(u)$ is a measure of the "urgency" of the task to be scheduled. It is given by:

$$D(u) = \begin{cases} L_u, \textit{ if there doesn't exist a } (u,v) \in E^S \\ \max_{e \in E^S}(D(v) + L_u - II \cdot \delta(e)), \textit{ otherwise} \end{cases}$$

where $e = (u, v)$. For an initiation interval II, $(II - D(u))$ gives the as late as possible (ALAP) schedule time of the task. Both these quantities can be calculated by a breadth first search of the DDG.

A path $p = \{e_1, \ldots, e_n\}$ is called a positive path, if $\forall e \in p$, e is a PSD. The Length of p is:

$$Length(p) = L_w + \sum_{(u,v) \in p} (L_u - II \cdot \delta(u, v)),$$

where L_w is the latency of the tail task in the positive path. For a task that is the head node of a positive path the above expression gives the depth of the task. A *maximal positive path*, MPP of a DDG, is a positive path p such that, for any other positive path $p' \subseteq E$, $Length(p) \geq Length(p')$. The MPP for a DDG is then given by:

$$MPP = max(D(u)), \forall u \in V$$

For a feasible schedule of a DDG with initiation interval II,

$$MPP \leq II.$$

Calculation of Memory Requirement Now let us consider the memory requirements of a pipeline schedule. We assume that the memory is reserved for the write set of a task as soon as it begins execution, and it remains reserved until the task which uses the data finishes execution. In other words, memory is reserved for some data as soon as the producer task begins execution and it is freed once the consumer task finishes execution. In a pipeline schedule the memory requirement is due to $ILDs$ and $LCDs$. $ILDs$ do not cross the boundary between two consecutive iterations of the steady state. All the data belonging to any ILD is produced and consumed within one iteration of the steady state. $LCDs$ cross the boundary between two iterations of the steady state. Depending on the distance (or δ) they might cross more than one boundary. Hence before an iteration of the staedy state can begin execution there is already some memory occupied by the LCD data which is given by :

$$Mem_{LCD} = \sum_{e \in LCD} e_{var} \times \delta(e)$$

Mem_{LCD} is the same at the beginning of each iteration of the steady state. Hence we need at least Mem_{LCD} memory for the pipeline design. The memory required during one iteration of the steady state is the maximum amount of memory occupied by the data items during execution, Mem_{exec}. This memory is both due to $ILDs$ and $LCDs$. The memory requirement of a pipelined design, MemReq is then given by:

$$MemReq = \max(Mem_{LCD}, Mem_{exec})$$

In the next section we discuss the list based scheduling algorithm.

4.1 List Based Scheduler

We use a list based scheduler for scheduling the DDG on the codesign architecture. The scheduler maintains three ready lists, one each for HW, SW and memory resource. The execution of a task can be divided in to *three states*. When a task is selected to be scheduled from either HW or SW ready list, it first goes in to *read state*. When the task has finished reading it goes in to *run state* and then in *write state* when its writing data to the shared memory. A task in the read and write states could cause a memory conflict with another task. The scheduler resolves conflicts by maintaining a ready list for the memory resource. A task is added to HW or SW ready list when all its predecessor tasks have been scheduled. When a task is selected to be scheduled on a particular resource, its goes into read state and is added to the memory ready list. A task on completion of its read operation runs on the appropriate resource and gets added to the memory ready list again when it goes into its write state.

The scheduler uses the same heuristic priority function to select a task from the three ready lists. The priority of a task to be selected depends on the following four properties in descending order :

1. 0-Mobility: The mobility of a task is given by the difference between its ALAP and ASAP times. The ASAP time may change during scheduling and its updated. The ALAP time of a task is constant for a given initiation interval. If a task has 0-Mobility then it must be scheduled at that

time. Otherwise the timing constraints will be violated.

2. Mobility: A task with lesser mobility is selected to be scheduled before a task with greater mobility. It is a well established heuristic which is known to produce good results.

3. Difference between number of read and write variables (or data items): The memory requirement of a schedule is given by the maximum memory occupied by the data items during one iteration of the steady state. A task which reads more variables than it writes would reduce the number of variables present in the memory. Hence it should be scheduled near its ASAP time. Alternatively a task which writes more variables than it reads should be scheduled near its ALAP time.

4. Number of Successors: A list scheduling algorithm performs better when it has more choice in the ready list. Hence a task whose completion adds more tasks to the ready list is selected.

A task with 0-mobility is always selected from the ready list. If no task has 0-mobility we use property 2 to select a task, and properties 3 and 4 (in that order) to break ties. In the next section we present the retiming heuristic.

4.2 Retiming Heuristic

Retiming increases the distance of a dependence and produces an equivalent DDG which satisfies the following condition:
Two graphs, $DDG = G(V, E, \lambda, \delta)$ and $DDG' = G(V, E, \lambda', \delta')$ are equivalent if, $\forall (u, v) \in E$, the following equation holds,

$$\lambda(v) - \lambda(u) + \delta(u, v) = \lambda'(v) - \lambda'(u) + \delta'(u, v)$$

We do retiming when we are unable to schedule a DDG in the given initiation interval, II. A successful schedule for a DDG can be obtained by decreasing the number of dependencies that constrain the schedule. By retiming we can transform a PSD into a FSD or NSD. The drawback of retiming is that it increases the memory requirement of the schedule. We can minimize this increase by using good heuristics to select the dependency to be retimed. But this is not enough. In order to produce an equivalent DDG other dependencies might need to be retimed. The increase in memory requirement due to these dependencies should also be minimized. During retiming we do not increase the distance of a dependence belonging to a recurrence. Also we ensure that no dependency has $\delta < 0$.

We do retiming in two steps. In the first step we heuristically select a dependency to be retimed. Increasing the distance of a dependence necessitates changing the λ and δ of other tasks and dependencies. In a DDG there might exist a number of sets of dependencies whose distance could be increased to obtain an equivalent retimed DDG. In step 2 we select the set of dependencies which on retiming result in the least increase in memory requirement. As a first step towards retiming we select a dependency to be retimed. The priority of a dependency to be retimed depends on its following four properties in decreasing order:

1. Dependency is a PSD: The primary objective of retiming is to reduce scheduling constraints in the DDG; and give the scheduler greater freedom in scheduling tasks on the resources. Only $PSDs$ constrain scheduling and therefore only $PSDs$ are retimed.

2. Dependency between tasks bound to heterogeneous resources: Increasing the distance of a dependency between tasks mapped to the same resource does not necessarily help the scheduler. Basically the two tasks have to be scheduled on the same resource and will be scheduled one after the other. On the other hand retiming a dependency between tasks mapped to different resources definitely gives more freedom to the scheduler.

3. Dependency whose predecessor task has a greater sum of height and depth ($H(u)+D(u)$): The sum of height ($H(u)$) and depth ($D(u)$) of a task gives the length of the positive path to which it belongs. Increasing the distance of a dependency whose predecessor task has a greater sum ($H(u) + D(u)$) reduces the length of a longer positive path in the DDG.

4. Dependency representing the least number of variables transferred: A secondary objective of retiming transformation is to minimize the increase in memory requirement of the DDG. Hence we select a dependency representing fewer variables being transferred.

We use property 1 to select dependencies to be retimed, and use properties 2 , 3 and 4 (in that order) to break ties. Given a dependency $e = (u, v)$ to be retimed we define the following four sets with respect to u:

V_c = { connected component to which u belongs }

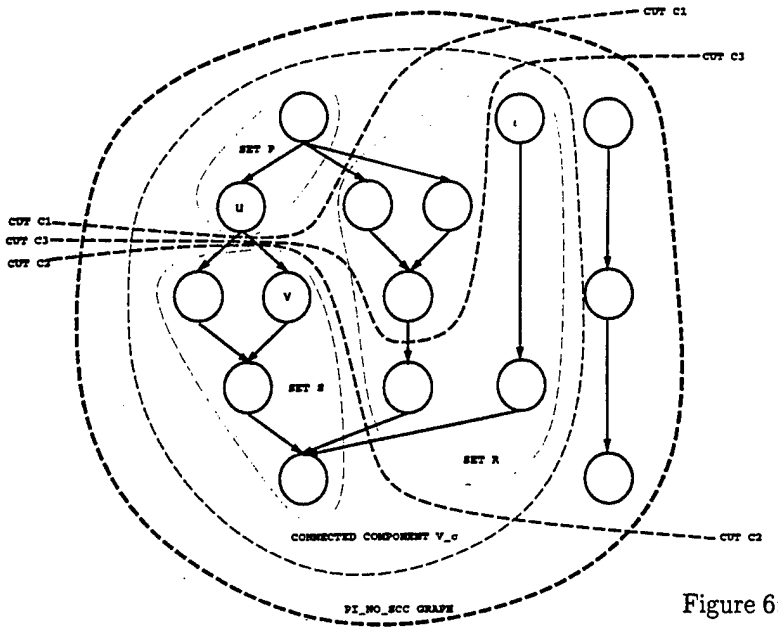


Figure 5: P , S and R sets during retiming of dependency (u,v)

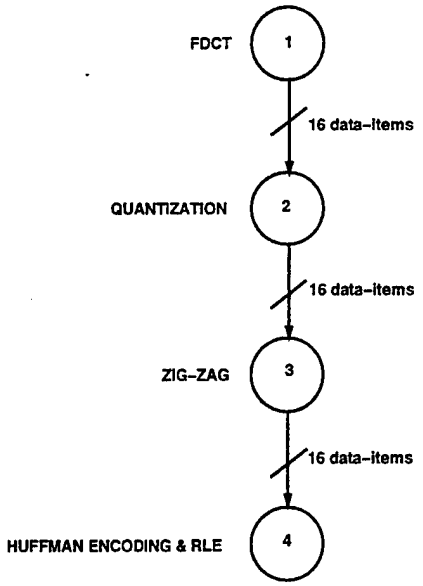


Figure 6: DDG for JPEG like Compression Algorithm

$$P = \{v \in V_c | \textit{there is a path from } v \textit{ to } u\ \} \cup \{u\}$$

$$S = \{v \in V_c | \textit{there is a path from } u \textit{ to } v\ \}$$

$$R = V_c - \{P \cup S\}$$

Figure 5 gives an illustration of the four sets. We can retime the dependency $e = (u, v)$ by the following three equations.

$$\lambda(u) = \lambda(u) + 1$$

$$\delta(u, x) = \delta(u, x) + 1, \forall x \in V \text{ such that } (u, x) \in E$$

$$\delta(x, u) = \delta(x, u) - 1, \forall x \in V \text{ such that } (x, u) \in E$$

Application of the three equations would result in an equivalent DDG. However the third equation decreases the distance of some dependencies. This can be avoided by increasing the λ of all tasks which are in P and increasing the δ of all dependencies whose predecessor task is in the set P and successor is in $R \cup S$. This is the *cutset c1* in Figure 5. Another way to retime is to increase the λ of all tasks in the set $P \cup R$ and increasing the δ of all dependencies whose predecessor is in $P \cup R$ and successor is in S. This is the *cutset c2* in Figure 5. However its possible that neither cutset c1 nor c2 give us a minimum increase in memory. We could obtain another *cutset c3* (see Figure 5) by partitioning the set R into P and S, so that the memory increase is minimized. We use a *simulated annealing* based partitioner. The cost function being minimized is defined as follows. For a cut $c_i = \{e_1, e_2, \ldots, e_n\}$, the cutsize cost is given by :

$$Cost = \sum_{j=1}^{n} var(e_j)$$

$var(e_j)$ is the number of variables across the dependency e_j. In the above cost function the sum gives us the extra memory required by the $LCDs$ after retiming. After partitioning R into P and S, we do retiming using the following two equations:

$$\forall u \in P, \lambda(u) = \lambda(u) + 1$$

$$\forall (u, v) \in E, u \in P, v \notin P, \delta(u, v) = \delta(u, v) + 1$$

5 Experimental Results

We demonstrate the effectiveness of the tool in codesign flow by considering the design of a JPEG [4] like compression algorithm. The DDG of the specification is shown in Figure 6. It consists of four tasks, Forward Discrete Cosine Transform (FDCT), Quantization, Zig-Zag and RLE and Huffman encoding. All the dependencies have $\delta = 0$ and the number of variables transfered across each dependency is 16. The memory read time is 16 ns and the memory write time is 24 ns respectively. The run times of the various tasks in SW and HW is shown in Table 1 [6]. Table 2 shows the comparison between throughput and

No.	Number of Tasks	Depth	Non-Pipeline		Pipeline			Speed-up (%)	Memory Incr. (%)
			Time (ns)	Memory	MII (ns)	II (ns)	Memory		
1	3	1	110	8	90	90	16	18	100
2	3	2	390	7	240	290	14	34.5	100
3	5	3	230	17	190	190	34	17.4	100
4	6	3	1410	30	1135	1170	75	17	150
5	8	5	750	170	600	600	190	20	11.7
6	8	7	890	10	730	730	20	18	100
7	8	7	740	10	425	470	20	36	100
8	8	7	890	5	465	580	20	35	300
9	10	4	1130	15	842	931	30	17.6	100
10	10	6	390	34	300	300	43	23	26
11	15	7	950	76	770	770	97	18.9	27.6
12	15	12	1290	52	860	860	66	33.3	26.9
13	20	7	1200	129	870	870	182	27.5	41
14	20	14	1150	96	1010	1010	104	12.2	8.3
15	50	6	6320	534	5640	5640	794	10.8	48.6

Table 3: Comparison between Non-Pipeline and Pipeline Implementations for Random Graphs

2, 4, 7, 8 and 9) we were not able to obtain pipeline schedules with MII as their initiation interval. This is because of the memory conflicts during scheduling and recurrences in the graph. Memory conflicts force the scheduler to defer a read or a write operation thereby increasing II. Dependencies belonging to recurrences are not retimed, hence they constrain the scheduler leading to an increase in II. The increase in memory requirement of a pipeline schedule is due to the extra memory that is required to store data items between two iterations of the steady state. It is quite common for the increase to be in the region of 100 to 300 percent. Speed-up due to pipelining was achieved for all graphs. For some graphs (rows 5, 10, 11, 12 and 13) a good speed-up was achieved with a low memory increment, thereby making them ideal candidates for pipelined implementation.

6 Conclusion

In this paper we have presented a pipeline scheduling technique for optimizing the throughput and memory requirements of HW-SW codesigns. The effectiveness of the technique was demonstrated by experimentation. This technique will be an integral part of a larger codesign tool now under development. Future work will involve extension of the technique to include general multiple ASIC architectures with different communication protocols.

References

[1] D.D. Gajski, N. Dutt, A. C-H Wu, S. Y-L lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[2] C.E. Leiserson and J.B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, Vol. 6, No. 1, pp. 5-35, 1991.

[3] F. Sánchez, *Loop Pipelining With Resource And Timing Constraints*, Ph.D. Dissertation, UPC Universitat Politècnica de Catalunya, Barcelona, Spain, October 1995.

[4] W.B. Pennebaker and J.L.Mitchell, *JPEG: Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993.

[5] N. Narasimhan, V. Srinivasan, M. Vootukuru, J. Walrath, S. Govindrajan, and R. Vemuri, "Rapid Prototyping of Reconfigurable Coprocessors", *Proceedings of the 1996 International Conferences on Application-Specific Systems, Architectures and Processors*, IEEE press, August 1996.

[6] J. Walrath, K. S. Chatha, R. Vemuri, N. Narasimhan and V. Srinivasan, "Performance Modeling and Tradeoff Analysis During Rapid

Prototyping", *Proceedings of the 1996 International Conferences on Application-Specific Systems, Architectures and Processors*, IEEE press, August 1996.

[7] R.K. Gupta and Giovanni De Micheli, "Hardware-Software Cosynthesis for Digital Systems", *IEEE Design and Test of Computers*, pp. 29-41, September 1993.

[8] R. Ernst, J. Henkel and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers", *IEEE Design and Test of Computers*, pp. 64-75, December 1993.

[9] T. Benner and R. Ernst, "A combined Partitioning and Scheduling Algorithm for heterogeneous Multiprocessor Systems", *Technical Report CY-96-2*, Institute of Computer Engineering, Technical University of Braunschweig, Germany.

[10] D.E. Thomas, J.K. Adams amd H. Schmit, "A Model and Methodology for Hardware-Software Codesign", *IEEE Design and Test of Computers*, pp. 6-15, September 1993.

[11] P.H. Chou, R.B. Ortega and G. Borriello, "The Chinook Hardware/Software Co-Synthesis System", *Proceedings of ISSS-95*, Cannes, France, September 13-15, 1995.

[12] P.Chou, E.A. Walkup and G. Borriello, "Scheduling for Reactive Real-Time Systems", *IEEE Micro*, pp 37-47, August 1994.

[13] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, S. Yee and A. Sangiovanni-Vincentelli, "Hardware-Software Codesign of Embedded Systems", *IEEE Micro*, pp 26-36, August 1994.

APPENDIX D :.

RECOD: A Retiming Heuristic To Optimize Resource And Memory Utilization In HW/SW Codesigns*

Karam S Chatha† *and Ranga Vemuri* ‡

Laboratory for Digital Design Environments
Department of ECECS
P.O. Box 210030
University of Cincinnati
Cincinnati, OH 45221–0030

SLIGHTLY REVISED VERSION OF THE PAPER ORIGINALLY ACCEPTED AT
EURODAC-97 (PAPER CODE D130).
THIS PAPER TAKES THE REVIEWERS COMMENTS IN TO ACCOUNT.

DATE-98 TOPIC 2 :HW/SW CODESIGN

All appropriate clearances for the publication of this paper have been obtained, and if accepted the authors will prepare the final manuscript in time for inclusion in the Conference Proceedings and will present the paper at the Conference.

(Karam S Chatha)

*This work was partially supported by the ARPA RASSP program and monitored by the Wright Lab, US-AF under contract number F33615-93-C-1316.

†Designated presenter, should the paper be accepted

‡Author for Correspondence,(513)-556-4784 (Voice), (513)-556-7326 (FAX), *Ranga.Vemuri@UC.EDU*

RECOD: A Retiming Heuristic To Optimize Resource And Memory Utilization In HW/SW Codesigns

Abstract

Hardware/Software designs of embedded systems are characterized by stringent performance constraints. Pipelined implementation of a design is an effective way for maximizing the performance of a design. In this paper we present a retiming heuristic to obtain pipelined schedules for hardware-software codesigns. The heuristic aims at maximizing the throughput of a resource constrained codesign while minimizing its memory usage. The effectiveness of the proposed technique is demonstrated by experimentation.

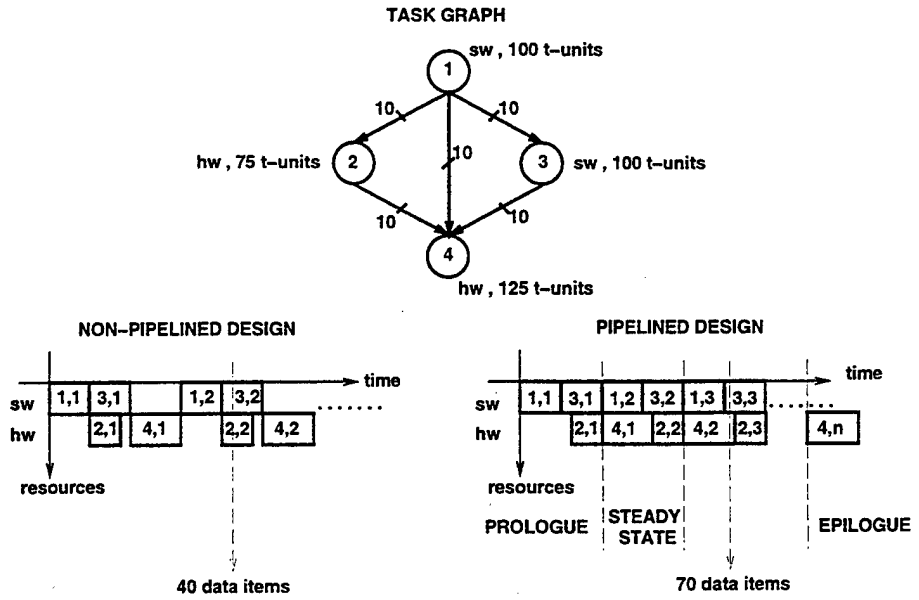


Figure 2: **Non-Pipelined and Pipelined Implementations of a Task Graph**

the task. The memory requirement of the implementation is the maximum memory used by one iteration of the loop (shown by the dotted line in the figure). This happens when tasks 2 and 3 execute in parallel. Task 2 needs memory for 20 data items and task 3 needs memory for 10 data items. Also at this point in time the variables transferred from task 1 to task 4 (10 data items) are also stored in the memory. Hence the maximum memory used by the implementation is for 40 data items. Now consider a pipelined implementation of the same task graph (lower right corner of the figure). A pipeline execution of a design can be divided into 3 parts. The first part which loads the pipeline is called the *prologue*. The second part is the *steady state* which is executed a several times. Finally the last part which down loads the pipeline is called the *epilogue*. As shown in the figure the execution of task 4 belonging to the first iteration of the loop is overlapped with execution of task 1 belonging to the second iteration. Once fully loaded the steady state completes one iteration of the loop every 200 t-units. A definite improvement over the previous design. The drawback is that the memory requirement has increased to 70 data items (shown by the dotted arrow line).

We implement pipelined designs by using retiming transformation. Retiming to generate pipelined design is considered a generalization [3] of the classical transformation introduced by Leiserson and Saxe [10]. A similar problem is the software pipelining problem [9] in code generation for VLIW architectures. *Given a task graph to be pipelined it can generally be retimed in more than one way. We need to select a retiming that gives us the least increase in memory requirements.* In this paper we present a *Retiming* heuristic for optimal resource and memory utilization in HW/SW *Codesigns* (RECOD).

In this paper we concentrate on the design of DSP applications. DSP applications have moderately simple algorithms and they demand high performance and throughput; thus necessitating search for efficient and

inexpensive implementations [13]. Besides many of these applications are loop oriented where a single block of code is executed a number of times on different set of data, thereby making them ideal candidates for pipelined implementation.

In this paper we assume that the SW processor and the ASIC in the codesign architecture are themselves non-pipelined with respect to task execution. We also assume that the pipeline schedule is rectangular in nature, that is a new iteration of the steady state does not begin before the previous one is over. In a non-rectangular schedule the execution of a task belonging to one iteration of the steady state overlaps with the execution of a task belonging to the next iteration.

The paper is organized as follows. In Section 2 we discuss previous work, in Section 3 we describe the graph representation and pipeline schedule, Section 4 presents RECOD, experimental results are in Section 5 and finally Section 6 concludes the paper.

2 Previous Work

The term "Retiming" was introduced by Leiserson and Saxe [10] when they used it to solve the problem of optimizing the throughput of synchronous circuitry. Retiming was used to describe the re-distribution of register delays between combinational blocks in a synchronous circuit. They developed an ILP formulation to solve the problem. Since then retiming transformation has been used extensively in logic synthesis [11], high level synthesis [15] [17], HW-SW codesign [18] and DSP applications [7] [8]. Pipelining is considered a generalization of the retiming problem in which circuit latency is allowed to increase by allowing a change in the production and consumption times of output and input signals respectively [3].

The term "Software Pipelining" was introduced by M. Lam [9]. She used it to describe a loop scheduling technique for code generation of VLIW processors. In software pipelining multiple iterations of the loop in various stage of their execution are in progress simultaneously. This description relates it very closely to pipelining in hardware systems. Since then a number of heuristic [1] [6] and ILP formulations [4] [12] have been proposed to solve the software pipelining problem. [16] gives a good comparison and survey of the techniques. [2] establishes a link between circuit retiming and software pipelining.

The work that comes closest to the paper is that of Sánchez presented in [17]. In that work, Sánchez has used a retiming heuristic in a high level synthesis tool that aims at obtaining pipelined designs with optimum throughput. The retiming heuristic retimes the head or tail dependency of the maximum positive path in a graph. In this paper we present a new retiming heuristic which optimizes both throughput and memory requirements of pipelined codesign applications. Our heuristic does retiming in two steps. In the first step it selects a dependency to be retimed which gives the maximum freedom to the scheduler. In the second step it selects the other dependencies (in addition to the first one) which on retiming result in an equivalent graph with the least increase in shared memory requirements. Experimental results show that our retiming strategy produces designs which use significantly lesser memory and operate at the optimum throughput rate.

3 Graph Representation and Pipeline Scheduling

Graph Representation The input specification is captured by an intermediate graph format called the *Data Dependency Graph* (DDG). It represents the tasks by vertices and the data dependencies between tasks by directed edges. The vertices have information about the task binding (HW or SW), HW run time and SW run time. The edges have information about the number of variables in a dependence. Since we are interested in pipelining the design, we associate with each vertex an iteration index (λ) and with each edge a dependency distance (δ). The iteration index of a task u, $\lambda(u)$ indicates that at the i^{th} iteration of the steady state, instance of task u belonging to the $(i + \lambda(u))$ iteration of the loop is executed. For example consider the pipelined design in Figure 2. In the first iteration of the steady state, instance of task 2 belonging to the second iteration of the loop is executed, hence $\lambda(task2) = 1$. Similarly $\lambda(task1) = 1, \lambda(task3) = 1$ and $\lambda(task4) = 0$. The dependence distance of an edge e, $\delta(e)$ indicates the distance of the dependence. In Figure 2 the data produced by task 1 at the i^{th} iteration of the steady state is consumed by task 4 at the $(i+1)^{th}$ iteration of the steady state. Hence the dependence distance of edge $(1,4)$ is $\delta(1,4) = 1$. Similarly $\delta(1,2) = 0, \delta(1,3) = 0, \delta(2,4) = 1$ and $\delta(3,4) = 1$. We now formalize the DDG representation as follows:

A DDG is a 4-tuple $DDG = G(V, E, \lambda, \delta)$, *where :*

- *V is the set of vertices. Each vertex $u \in V$ represents a task. For each task $u \in V$ we have the following information available to us :*
  - u_{bind} *: The binding of the task, that is whether its going to be implemented in HW or SW.*
  - u_{sw} *: The SW runtime of the task for a particular input data on the general purpose processor.*
  - u_{hw} *: The HW runtime of the task if it were to be implemented as an ASIC for the same input data.*
- *E is the set of directed edges. Each $e = (u, v) \in E$ represents a data dependence between tasks u and v. Every edge has information about the number of variables (e_{var}) represented by the dependence.*
- *λ and δ are two mappings, $\lambda : V \rightarrow \mathbb{N}$ and $\delta : E \rightarrow \mathbb{N}$, representing the iteration index (λ) and the number of iterations traversed by the dependence (δ), also called dependence distance. $\mathbb{N}$ is the set of natural numbers.*

Initially, $\forall u \in V$, $\lambda(u) = 0$. Notice that the representation has no control flow constructs; it is strictly data flow.

Theoretical Upper Bound on Throughput Given a DDG there exists a theoretical upper bound on the throughput of a pipeline schedule of the graph [17]. It is called the *maximum execution throughput (MaxTh)* and it gives the maximum number of iterations of the steady state in one time unit. The reciprocal of $MaxTh$ is called the *minimum initiation interval (MII)*. For a particular pipeline implementation the *initiation interval, II*, is the time taken for one iteration of the steady state. For example in Figure 3, the

pipelined implementation has $II = 200$ t-units. The MII is limited by two factors. Firstly the number of resources (HW or SW) limit MII. This is called the *resource constrained MII, ResMII.* Consider again the example shown in Figure 2. The task graph has two tasks 1 and 3 bound to SW. Hence we need at least 200 t-units to complete the execution of task 1 and 3. Similarly we need at least 200 t-units to complete execution of tasks 2 and 4 in HW. The $ResMII_i$ due to a resource i is given by the ratio of the sum of the latencies of all the tasks executing on the resource i by the total number of instances of resource i [17]. The latency of a task u, L_u, is the total execution time of the task. It is the sum of the task's read time, execution time on the particular resource that its been bound to and write time. The read (write) time of a task is the product of the number of variables read (written) by the task and the memory read (write) time. Hence we have,

$$L_u = u_i + u_{rdtime} + u_{wrtime} \quad where,\ u_i = \begin{cases} u_{sw} & if\ u_{bind} = sw \\ u_{hw} & if\ u_{bind} = hw \end{cases}$$

Since the codesign architecture has only one HW and one SW resource, we can calculate $ResMII_{HW}$ and $ResMII_{SW}$ as the sum of latencies of all tasks bound to HW and SW respectively. $ResMII$ for a DDG is the maximum of all the $ResMII_i$, therefore we have $ResMII = max(ResMII_{HW}, ResMII_{SW})$. Secondly recurrences or cycles in a task graph also limit MII. This is called the *recurrence constrained* MII, $RecMII$. Let us assume that in Figure 2, the data produced by task 4 in i^{th} iteration of the loop is consumed by task 1 in the $(i+1)^{th}$ iteration, that is let us add an edge $e = (4,1)$ with $\delta(4,1) = 1$ to the task graph. In such a case the schedule shown in the figure becomes invalid. This is because now we cannot overlap the execution of task 1 and task 4. Infact any schedule of the graph now takes at least 325 t-units. The $RecMII_r$ for a recurrence r, is given by the ratio of the sum of the latencies of the tasks in the recurrence to the sum of the weights (δ) of all the dependencies in a recurrence [17]. A graph may have more than one cycle, and $RecMII$ is then the maximum of the $RecMII_r$ due to each one of them, that is $RecMII = max(RecMII_r)$, for all the recurrences r in the DDG. The MII is then the maximum of $ResMII$ and $RecMII$. That is,

$$MII = \max(ResMII, RecMII) \Rightarrow MaxTh = \frac{1}{(\max(ResMII, RecMII))}$$

Pipeline Schedule The pipeline schedule of a task graph is characterized by its initiation interval II. The schedule is an assignment of start times to tasks, $S(u)$, such that for all tasks u in the graph $0 \leq S(u) \leq (II-1)$. For a dependency (u,v), the schedule time of u and v must honor the data dependence, that is

$$S(v) + \delta(u,v) \cdot II \geq S(u) + L_u \Rightarrow S(v) \geq S(u) + L_u - \delta(u,v) \cdot II$$

As we will see in the next paragraph not all dependencies constrain a pipeline schedule. The dependencies which do not constrain a schedule can be ignored during scheduling. We obtain a pipeline schedule by scheduling [5] and retiming in an iterative manner as shown in Figure 3. We calculate the MII, and try scheduling the DDG for MII. However due to constraining dependencies we may not be able to schedule the DDG in MII. If we can't we retime the DDG and try again. *The objective of retiming is to reduce the number of schedule constraining dependencies.*

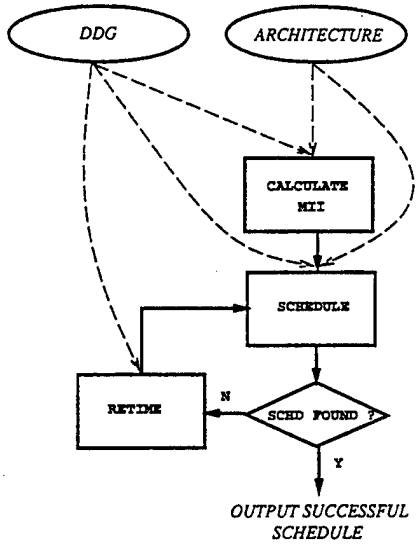


Figure 3: **Pipeline Scheduling by Iterative Retiming**

Schedule Constraining Dependencies Depending on whether $\delta(u,v)$ is equal or greater than zero a data dependency (u,v) may or may not constrain a pipeline schedule. A dependency with $\delta(u,v) = 0$ constrains a pipeline schedule. This is because now $S(v) \geq S(u) + L_u$ is strictly positive. Essentially a data dependence with $\delta(u,v) = 0$ implies that the data produced by the predecessor task u is consumed by the successor task v in the same iteration of the steady state and hence it constrains the schedule. Such a dependency is called a *positive scheduling dependency (PSD)* [17] or *intra loop dependency (ILD)*. A dependency (u,v) with $\delta(u,v) > 0$ gives us two cases. First consider a dependency dependency with $\delta(u,v) > 0$ and $L_u - II \cdot \delta(u,v) < -(II-1)$. Such a dependency does not constrain a pipeline schedule since for all values of $S(u)$ and $S(v)$ the data dependence is satisfied, that is

$$\text{If } \delta(u,v) > 0 \text{ and } L_u - II \cdot \delta(u,v) < -(II-1) \text{ then,}$$

$$S(v) \geq S(u) + L_u - \delta(u,v) \cdot II, \forall S(u), S(v) \in [0, II).$$

Such a dependency is called a *free scheduling dependency (FSD)* [17]. Now consider a dependency with $\delta(u,v) > 0$ and $-(II-1) \leq L_u - II \cdot \delta(u,v) \leq 0$. Such a dependency is called a *negative scheduling dependency (NSD)* [17] and it will constrain a pipeline schedule under two conditions. Firstly if the pipeline schedule is non-rectangular then the $NSDs$ would constrain the schedule. Secondly if the resources on which tasks u and v are executing are themselves pipelined then $NSDs$ would constrain the schedule. Since neither of these two conditions are true in our case $NSDs$ do not constrain the pipeline schedule. $FSDs$ and $NSDs$ together are called *loop carried dependencies (LCDs)* since they represent a data dependence between tasks executing in different iterations of the steady state. Hence for a given initiation interval II, the set of schedule constraining dependencies, E^S is set of $PSDs$ in the DDG, that is

$$E^S = \{(u,v) \in E | \delta(u,v) = 0\}$$

The initiation interval II of a pipeline schedule is constrained by the length of the *maximum positive path (MPP)* in the DDG. A path $p = \{e_1, \ldots, e_n\}$ is called a positive path, if $\forall e \in p$, e is a schedule constraining dependency. The Length of p is:

$$Length(p) = L_w + \sum_{(u,v)\in p} (L_u),$$

where L_w is the latency of the tail task in the positive path. A *maximal positive path*, MPP of a DDG, is a positive path p such that, for any other positive path $p' \subseteq E$, $Length(p) \geq Length(p')$. For a feasible schedule of a DDG with initiation interval II,

$$Length(MPP) \leq II.$$

Hence during retiming we should try to reduce the number of schedule constraining dependencies which to a longer positive path. Before we present the retiming algorithm in the next section, we discuss the memory requirements of a pipeline schedule in the following paragraph.

Calculation of Memory Requirement We assume that the memory is reserved for the write set of a task as soon as it begins execution, and it remains reserved until the task which uses the data finishes execution. In other words, memory is reserved for some data as soon as the producer task begins execution and it is freed once the consumer task finishes execution. In a pipeline schedule the memory requirement is due to $ILDs$ ($PSDs$) and $LCDs$ ($FSDs$ and $NSDs$). $ILDs$ do not cross the boundary between two consecutive iterations of the steady state. All the data belonging to any ILD is produced and consumed within one iteration of the steady state. $LCDs$ cross the boundary between two iterations of the steady state. Depending on the distance (or δ) they might cross more than one boundary. Hence before an iteration of the steady state can begin execution there is already some memory occupied by the LCD data which is given by :

$$Mem_{LCD} = \sum_{e\in LCD} e_{var} \times \delta(e)$$

Mem_{LCD} is the same at the beginning of each iteration of the steady state. Hence we need at least Mem_{LCD} memory for the pipeline design. The memory required during one iteration of the steady state is the maximum amount of memory occupied by the data items during execution, Mem_{exec}. This memory is both due to $ILDs$ and $LCDs$. The memory requirement of a pipelined design, $MemReq$ is then given by:

$$MemReq = \max(Mem_{LCD}, Mem_{exec})$$

As we see by the above discussion Mem_{LCD} is a lower bound on the memory requirement of a pipeline schedule. During retiming we convert a schedule constraining dependency (ILD) in to a LCD which does not constrain the schedule, thereby increasing Mem_{LCD}. *Therefore during retiming we should try to reduce the increase in* Mem_{LCD}.

Each task in the DDG is bound to a unique resource. Hence $ResMII$ is an achievable lower bound. In other words we should be able to schedule the DDG in MII time when the binding is known (and $RecMII <$

ResMII). The general case where binding is unknown increases the complexity of the scheduler. However, the retiming heuristic should work equally well in the general case.

4 RECOD: Retiming Heuristic for HW/SW Codesigns

We do retiming when we are unable to schedule a DDG in the given initiation interval, II. A successful schedule for a DDG can be obtained by decreasing the number of dependencies that constrain the schedule. By retiming we can transform a PSD into a FSD or NSD ($LCDs$) by increasing the dependence distance (δ). LCDs do not constrain an iteration of the loop. During retiming we ensure that no dependency has $\delta < 0$. Also retiming should produce an equivalent DDG. Two graphs, $DDG = G(V, E, \lambda, \delta)$ and $DDG' = G(V, E, \lambda', \delta')$ are equivalent if, $\forall (u, v) \in E$, the following equation holds,

$$\lambda(v) - \lambda(u) + \delta(u, v) = \lambda'(v) - \lambda'(u) + \delta'(u, v)$$

Retiming produces a DDG with tasks belonging to different iterations. In other words dependence retiming helps in pipelining a DDG.

The drawback of retiming is that it increases the memory requirement of the schedule. Since we now have tasks belonging to different iterations executing at the same time, we need more shared memory to store data between successive iterations of the steady state. We can minimize this increase by using good heuristics to select the dependency to be retimed. But this is not enough. In order to produce an equivalent DDG other dependencies might need to be retimed. The increase in shared memory requirement due to these dependencies should also be minimized. Hence RECOD does retiming in two steps. In the first step it heuristically selects a dependency to be retimed. Increasing the distance of a dependence necessitates changing the λ and δ of other tasks and dependencies. Decreasing the δ of a dependence is likely to change it in to a PSD. Hence during retiming we only increase the distance of the dependencies. In a DDG there might exist a number of sets of dependencies whose distance could be increased to obtain an equivalent retimed DDG. In step 2 we select the set of dependencies which on retiming result in the least increase in shared memory requirement.

The distance of a dependency belonging to a recurrence in the DDG cannot be increased without decreasing the distance of any other dependency. Hence during retiming we do not increase the distance of a dependence belonging to a recurrence. A dependence not belonging to a recurrence can however be retimed without decreasing the distance of another dependence.

4.1 RECOD Step 1: Heuristic To Select A Dependency For Retiming Transformation

As a first step towards retiming we select a dependency to be retimed. The priority of a dependency to be retimed depends on its following four properties in decreasing order:

1. Dependency is a PSD.
   The primary objective of RECOD is to reduce scheduling constraints in the DDG; and give the scheduler greater freedom in scheduling tasks on the resources. Only $PSDs$ constrain scheduling. Hence the dependency to be retimed should be a PSD, and not a NSD or FSD.

2. Dependency between tasks bound to heterogeneous resources.
   As mentioned above the main objective of the retiming heuristic is reduce scheduling constraints in the graph. Increasing the distance of a dependency between tasks mapped to the same resource does not necessarily help the scheduler. Basically the two tasks have to be scheduled on the same resource and will be scheduled one after the other. On the other hand retiming a dependency between tasks mapped to different resources definitely gives more freedom to the scheduler.

3. Dependency whose predecessor task belongs to a longer positive path.
   As discussed in the previous section the positive paths limit the II of a pipeline schedule. Increasing the distance of a dependency whose predecessor task belongs to a longer positive path helps in obtaining a pipeline schedule with smaller II and therefore higher throughput.

4. Dependency representing the least number of variables transferred.
   A secondary objective of retiming transformation is to minimize the increase in memory requirement of the DDG. Increasing the distance of a dependency with more variables definitely results in a larger increase in memory requirement. Hence we select a dependency representing fewer variables being transferred.

We use property 1 to select dependencies to be retimed, and use properties 2 , 3 and 4 (in that order) to break ties.

4.2 RECOD Step 2: Partitioning To Minimize Increase In Memory Requirement During Retiming

The primary objective of retiming is to give the scheduler greater freedom. This is achieved by the heuristic described above. We now select the set of dependencies which give us the least increase in memory requirement. Given a dependency $e = (u, v)$ to be retimed we define the following four sets with respect to u:

$$V_c = \{\textit{connected component to which } u \textit{ belongs }\}$$

$$P = \{v \in V_c | \textit{there is a path from } v \textit{ to } u \} \cup \{u\}$$

$$S = \{v \in V_c | \textit{there is a path from } u \textit{ to } v \}$$

$$R = V_c - \{P \cup S\}$$

Figure 4 gives an illustration of the four sets. We can retime the dependency $e = (u, v)$ by the following three equations.

$$\lambda(u) = \lambda(u) + 1$$

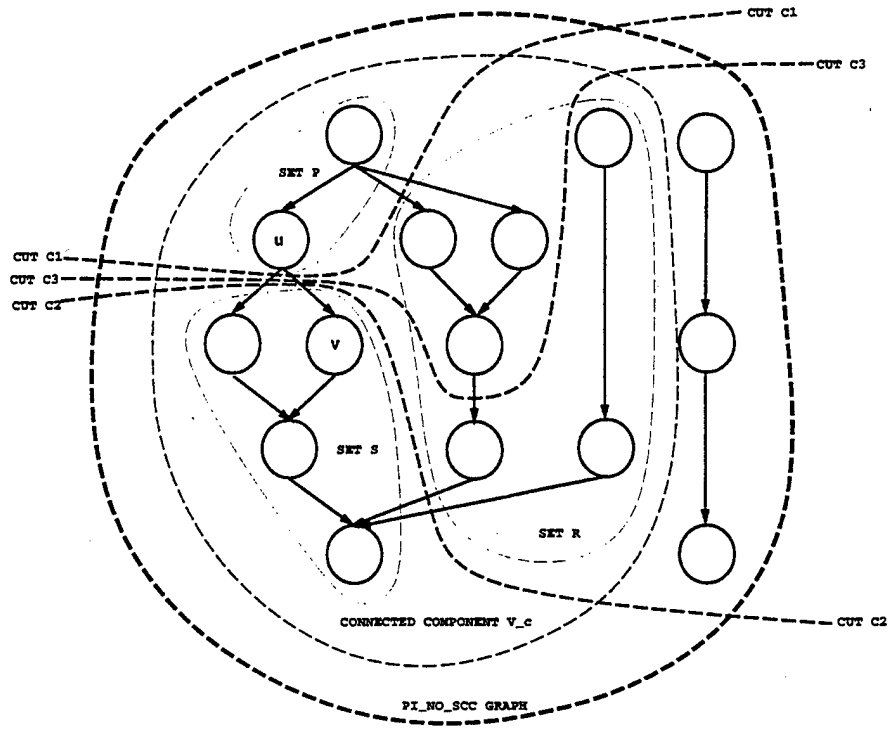


Figure 4: P, S **and** R **sets during retiming of dependency** *(u,v)*

$$\delta(u,x) = \delta(u,x) + 1, \forall x \in V \text{ such that } (u,x) \in E$$

$$\delta(x,u) = \delta(x,u) - 1, \forall x \in V \text{ such that } (x,u) \in E$$

Application of the three equations would result in an equivalent DDG. However the third equation decreases the distance of some dependencies. This can be avoided by increasing the λ of all tasks which are in P, that is $\forall u \in P, \lambda(u) = \lambda(u) + 1$. Now to obtain an equivalent DDG we need to increase the δ of all dependencies whose predecessor task is in the set P, but successor isn't, that is $\forall (u,v) \in E, u \in P, v \notin P, \delta(u,v) = \delta(u,v) + 1$. This is the *cutset c1* in Figure 4. Another way to retime without decreasing the δ of any dependence is as follows, $\forall u \in \{P \cup R\}, \lambda(u) = \lambda(u) + 1$ and $\forall (u,v) \in E, u \notin S, v \in S, \delta(u,v) = \delta(u,v) + 1$. This is the *cutset c2* in Figure 4. However it is possible that neither cutset $c1$ nor $c2$ might give us a minimum increase in memory. We could obtain another *cutset c3* (see Figure 4) by partitioning the set R into P and S, so that the memory increase is minimized. We use a *simulated annealing* based partitioner. The cost function being minimized is defined as follows. For a cut $c_i = \{e_1, e_2, \ldots, e_n\}$, the cutsize cost is given by :

$$Cost = \sum_{j=1}^{n} var(e_j)$$

$var(e_j)$ is the number of variables across the dependency e_j. In the above cost function the sum gives us the extra memory required by the $LCDs$ after retiming. During partitioning we ensure that if a task u is in partition P (S) then all its predecessors (successors) are also in partition P (S). After partitioning set

Algorithm *RECOD: Retimes the DDG*

Input : DDG

Output : *Retimed DDG with less number of PSDs*

Begin

$DDG_{no_scc} = remove_scc(DDG)$

$edge_{(u,v)} = heuristic_select(DDG_{no_scc})$

if $(edge_{(u,v)} = 0)$ **then return**$(DDG, failure)$

$V_c = \{$*connected component to which u belongs*$\}$

$S = \{v \in V_c |$ *there is a path from u to v*$\}$

$P = \{v \in V_c |$ *there is a path from v to u*$\} \cup \{u\}$

$R = V_c - \{S \cup P\}$

partition(R,P,S)

for *each* $x \in V_c$

  if $(x \in P)$ **then** $\lambda(x) = \lambda(x) + 1$ **endif**

endfor

for *each* $(x, y) \in E_c$

  if $(x \in P$ *AND* $y \in S)$ **then** $\delta(x, y) = \delta(x, y) + 1$ **endif**

endfor

$copy_changes(DDG_{no_scc}, DDG)$

return$(DDG, success)$

end

Figure 5: **RECOD: Algorithm**

R in to sets P and S we do retiming using the following two equations:

$$\forall u \in P, \lambda(u) = \lambda(u) + 1$$

$$\forall (u, v) \in E, u \in P, v \notin P, \delta(u, v) = \delta(u, v) + 1$$

4.3 RECOD: Algorithm

The algorithm to do retiming transformation is shown in figure 5. A brief explanation of the functions used in the algorithm are as follows. The function *remove_scc()* replaces every strongly connected component, scc_i (or recurrence) in the DDG with a single task u_{scc_i}. It returns a new graph DDG_{no_scc}. All the dependencies that are part of a recurrence scc_i are not present in DDG_{no_scc}. All the dependencies that are "to" and "from" any task in the scc_i are now from the single task u_{scc_i}. We use DDG_{no_scc} for retiming. By removing all the scc tasks and dependencies we ensure that no dependency belonging to a recurrence is retimed; although the λ of all the tasks belonging to a recurrence might be increased. The changes are reflected in the original DDG by the function *copy_changes()*. The function *heuristic_select()* heuristically selects a dependency to be retimed (see section 5.1). The function *partition()* as the name

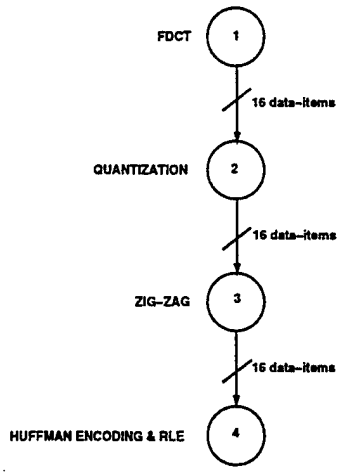


Figure 6: **DDG for JPEG like Compression Algorithm**

id.	Task	SW time(ns)	HW time(ns)
1	FDCT	371300	8400
2	Quant.	7560	600
3	ZigZag	1630	400
4	RLE & Huff.	18480	884000

Table 1: **SW and HW run times for various JPEG tasks**

suggests partitions R between P and S (see section 5.2). The two *for-loops* do the retiming. The first one increases the λ of all tasks $u \in P$. The second one increases the δ of all dependencies $(u, v), u \in P, v \in S$.

5 Experimental Results

To demonstrate the effectiveness of the retiming heuristic in HW/SW codesign, we consider the design of a JPEG [14] like compression algorithm. The DDG of the specification is shown in Figure 6. It consists of four tasks, Forward Discrete Cosine Transform (FDCT), Quantization, Zig-Zag and RLE and Huffman encoding. All the dependencies have $\delta = 0$ and the number of variables transfered across each dependency is 16. The respective run times of the various tasks in SW and HW is shown in Table 1 [19]. Table 2 shows the estimated throughput and memory requirements for various bindings of the tasks. Columns two to five give the bindings of the tasks. The sixth and seventh columns have the run time and memory requirement of the non-pipeline design of the application. The eighth column gives the MII of the pipeline implementation. Columns nine and ten give the achieved II and the memory requirement of the pipeline implementation. The speed-up and increase in memory requirement due to pipeline implementation are in columns eleven and twelve respectively. In the table we have exhaustively bound all the tasks to SW and HW. Since we have four tasks, we have sixteen rows in the graph. The results show that we were always able to schedule the DDG in MII time. We can achieve a speed-up of upto 1.6 (row 15). The maximum

[6] R.A. Huff, "Lifetime Sensitive Modulo Scheduling", *Proceedings of the '93 SIGPLAN conference on Programming Language Design and Implementation*, pp 258-267, June 1993.

[7] S. Huang and J. Rabaey, "Maximizing the Throughput of High Performance DSP Applications using Behavioral Transformations", *Proceedings of EDAC-ETC-EUROASIC '94*, pp 25-40, March 1994.

[8] L. Jeng and L. Chen, "Rate-Optimal static scheduling for recursive DSP algorithms by retiming and unfolding", *International Journal of Electronics*, 1992, Vol. 73, No. 4, pp 687-701.

[9] M. Lam, "Software Pipelining: An effective scheduling technique for VLIW Machines", *ACM SIGPLAN*, 1988.

[10] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry", *Algorithmica*, vol. 6, no. 1, pp. 5-35, 1991.

[11] S. Malik, K.J. Singh, R.K. Brayton and A. Sangiovanni-Vincentelli, " Performance Optimization of Pipelined Logic Circuits Using Peripheral Retiming and Resynthesis"; *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol 12, No. 5, May 1993.

[12] Q. Ning and G.R. Gao, "A Novel framework of Register Allocation for Software Pipelining" , *Conference Record* 20^{th} *Annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pp 29-42, Jan 10-13, 1993.

[13] N. Narasimhan, V. Srinivasan, M. Vootukuru, J. Walrath, S. Govindrajan, and R. Vemuri, "Rapid Prototyping of Reconfigurable Coprocessors", *Proceedings of the 1996 International Conferences on Application-Specific Systems, Architectures and Processors*, IEEE press, August 1996.

[14] W.B. Pennebaker and J.L.Mitchell, "JPEG: Still Image Data Compression Standard", *Van Nostrand Reinhold*, 1993.

[15] M. Potkonjak and J. Rabaey, "Optimizing Resource Utilization Using Transformations", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol 13. No. 3, March 1994.

[16] J. Ruttenberg, G.R. Gao, A. Stoutchinin and W. Lichtenstein, "Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler", *ACM SIGPLAN NOTICES*, May 1996.

[17] F. Sánchez, "Loop Pipelining With Resource And Timing Constraints", Ph.D. Dissertation, UPC Universitat Politècnica de Catalunya, Barcelona, Spain, October 1995.

[18] M. Sheliga, N.L. Passos and E.H. Sha, "Fully Parallel Hardware/Software Codesign For Multi-Dimensional DSP Applications", *Proceedings of* 4^{th} *International Workshop on Hardware/Software Co-Design (Codes/CASHE '96)*, March 1996.

[19] J. Walrath, Karam S. Chatha, R. Vemuri, N. Narasimhan and V. Srinivasan, "Performance Modeling and Tradeoff Analysis During Rapid Prototyping", *Proceedings of the 1996 International Conferences on Application-Specific Systems, Architectures and Processors*, IEEE press, August 1996.

APPFNDIX E:

Hardware/Software CoSynthesis: Multiple Constraint Satisfaction and Component Retrieval*

R. Miller H. Carter K. Davis[1]
Electronic Design Automation Research Center
University of Cincinnati
Cincinnati, OH 45221-0030
{rmiller, hcarter, kcd}@ece.uc.edu

S. Venkatesan
Intel Corporation
RN4-40, 2200 Mission Clg. Blvd.
Santa Clara, CA 95052
satish@scdt.intel.com

Abstract

Hardware/software CoSynthesis is a complex process that involves transforming a high-level system specification to an implemented hardware/software system that meets the specification constraints. One phase of the CoSynthesis process is described here: partitioning the specification into components and binding them to hardware/software resources. Partitioning requires an effective means to explore the design space; challenges include (1) supporting constraint-driven retrieval and (2) evaluating candidate solutions considering the interaction of multiple constraints. The CoSynthesis Tool described here assigns scores to candidate solutions using multiple design constraints, but rather than the simple sum approach predominant in CoSynthesis research, it uses a vector of rank data that does not require that equal weight be given to all criteria. Our results to date show that not only can we can process a scaleable, selectable set of design constraints, but when compared with a 2 constraint Fidducia-Matheyses (FM) approach, we achieve better results. The flexible component retrieval is accomplished using our database system; the database is unique for three reasons: (1) it uses a hardware description language as the basis for its conceptual model, (2) it allows flexible, ad hoc querying over designs, and (3) it uses a fine granularity of component modeling to enable detailed search conditions required by the CoSynthesis Tool.

1. Introduction

Hardware/software CoDesign and CoSynthesis can be characterized as a binding problem: binding components from a database to functional specifications in order to create a hardware/software system that carries out the desired functionality and meets performance constraints. The CoDesign methodology used in our research is embodied in the hardware/software CoDesign and CoSynthesis project called COMET [Vem94]. The general goal of COMET is to transform high level system specifications into application specific electronic signal processing modules using a hardware/software CoSynthesis process and to produce working hardware within a two week time period. HW/SW CoDesign/CoSynthesis is assumed to be the requisite approach for reducing the development cycle [Gaj94] and time to market. Current time to market for a complex HW/SW system is approximately 18 months [Keu94].

An abstract representation of the major COMET system components is given in Figure 1. A user supplies a system specification that is divided into modules, matched to component specifications, and then allocated to either hardware or software synthesis processes. The CoSynthesis process is iterative; alternate bindings are used to satisfy constraints such as performance and area requirements. The CoSynthesis Tool issues requests to the design database using qualifications on design properties, and the query processor determines the set of design objects that subsume the request. In other words, a query is a module description, and any modules in the database that have at least the desired functionality (possibly additional functionality) are returned. The CoSynthesis Tool analyzes candidate solutions and determines the best assignment of resources to hardware and software using an iterative binding algorithm. The hardware and software specifications are processed by hardware and software synthesis tools, then integrated to form a system that satisfies the initial specifications. The end result of these transformations is an application specific hardware design that can be fabricated along with the embedded software that will be executed on the manufactured hardware. The shaded portion of Figure 1

* Partially supported by NSF Grant IRI-9210200 and ARPA's RASSP Technology Program, contract F 33615-93-C-1316.
[1] Author for correspondence. Phone: (513) 556-2214. Fax: (513) 556-7326

highlights the subsystems described in this paper, the CoSynthesis Tool and the design database.

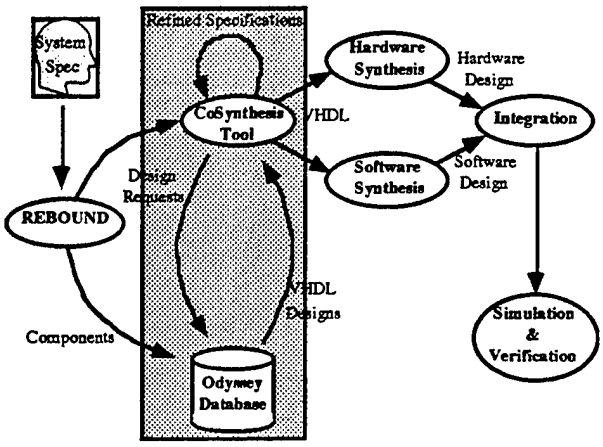


Figure 1. Application Environment.

2. The CoSynthesis Tool

The goal of the HW/SW CoSynthesis tool is to allocate hardware and software resources for the modules given in a high-level system specification. Input to the CoSynthesis Tool specifies the system functionality and performance constraints levied on the system by the designer. They can specify (but are not limited to) the final design's size, weight, power consumption, heat dissipation, and speed. The output of the CoSynthesis tool consists of bindings of modules to resources. The resources come from a pre-defined component database. It is the interplay of their attributes (size, weight, power, etc.) with particular bindings of resources to actions that determines how well the final design meets the performance constraints [Mil95]. In this paper, our preliminary implementation produces a VHDL configuration body as output. VHDL uses configuration bodies to specify bindings between components within a design and their implementation in a VHDL library of components. Extensions to this research have the goal of producing a configuration body and an updated architecture reflecting hardware and software resource allocations.

The relationship of our CoSynthesis algorithm and algorithms used for traditional hardware partitioning is described in Section 2.1. Our algorithm is proposed in Section 2.2.

2.1 Related Work

Iterative techniques such as Simulated Annealing (SA), Kernighan-Lin (KL), Fiduccia-Mattheyses (FM), and Genetic Algorithms (GA) are commonly used in hardware partitioning [She94] and have been in use for a decade or more [Bha94]. Hardware partitioning provides a means for breaking a system design up into smaller, more manageable pieces based primarily on the number of communication channels between the pieces. Hardware partitioning is not limited to one level of design abstraction or even application area. It can be used to facilitate design packaging [Bha94], design layout [Bha94], simulation and test [Cha94], Rapid Prototyping [Cha94], and logic minimization [Con94].

Given an initial partitioning of a system into two halves, iterative techniques move one circuit component (node), or pairs of nodes, between the partitions in an effort to minimize a single constraint or a pair of constraints. At the core of these algorithms is the manner in which they select the "best node" within the system graph to move between partitions. These techniques are a natural extension for HW/SW CoSynthesis and are the core iterative technique of many CoDesign or CoSynthesis approaches [Ben93] [Car96] [Gaj94] [Gup93] [Hen96] [Yeh95].

In the HW/SW CoSynthesis context, the hardware partitions become software and hardware partitions respectively. The movement of system nodes between the two is accomplished by rebinding the node's physical implementation from hardware to software or vice-versa. However, while cutset minimization remains a meaningful design constraint, area balancing does not. Further, one of the COMET project's goals is to facilitate additional design constraints in the CoSynthesis process. The iterative improvement algorithms are limited by their ability to readily add additional design constraints due to their manner of selecting the "best node" to move between partitions.

The two most common hardware partitioning algorithms differ in how they select the "best node" to move. The Fidducia-Matheyses method (FM) [She94] for hardware partitioning starts from an initial partitioning of the system graph. It proceeds by rank ordering all the tasks in the graph based on how moving a task from one chip to the other impacts the overall inter-chip communication (cutset). Next, the rank ordered list is stepped through and the algorithm selects the first task from the list that reduces the cutset and does not violate a predetermined size balance (usually set at 40-60%) between the two chips. This task is then moved to the other partition and the ranked list is updated. This process repeats until all tasks have been moved. The history of all task moves is examined to find the point in the process where the cutset is minimized.

The Ratio Cut method [Wei91] [Cha94] evaluates the tasks based on the following equation

$$C_{AA'} = \sum_{i \in A} \sum_{j \in A'} c_{ij} \quad R_{AA'} = \frac{C_{AA'}}{(|A| \times |A'|)}$$

where A and A' are the two hardware partitions and C is the cutset between the partitions. This equation takes into account the number of communication lines and the relative sizes of the two partitions. Once all the tasks have been evaluated, the task with the smallest value, R, is selected for movement.

The FM method may be extended for additional constraints, but either each constraint must be expressed as a range or the task ranking must be based on an equation that incorporates the results of the constraint evaluations as a simple sum. The first method is imprecise; the second mixes incomparable attributes. The Ratio Cut method suffers from the same restrictions.

Our algorithm improves on the iterative improvement technique by selecting the "best node" for rebinding rather than the first node that is acceptable, as well as allowing additional constraints to be added easily to the evaluation process. Our work is primarily influenced by techniques from hardware partitioning, but we have taken an approach similar to that of the DESTINATION project [Mar96] for assigning tasks to processors in complex computer systems. They consider multiple constraints with user-defined weights combined into a single objective function, similar to our approach.

2.2 CoSynthesis Algorithm

The new algorithm, called SCOREBOARD, has its roots in the FM method. Our algorithm maintains separate, rank-ordered lists for each node that may be rebound for each constraint specified by the system specification. Each constraint specifies the scalar value of one dimension of a ranking vector for that node. The "best node" to move is selected by choosing the node with the smallest vector from the set of possible candidates to rebind. After preliminary system definitions in Sections 2.2.1 and 2.2.2, the algorithm is described in Section 2.2.3.

2.2.1 Component Database

During CoSynthesis, all nodes from the system are bound to a specific implementation from a database or library of hardware and software components. The component library, L, consists of components, $l_{j,k}$, where:

- j specifies the class or functionality of the library component, and
- k specifies the particular implementation for the component.

Using VHDL as the design language, VHDL entity/architecture pairs represent the j's and k's. Additionally, for each $l_{j,k}$ component there exists a set of performance attributes, p_i, and a set of functions, f_j. Sample p_i's include size, cost, weight, and area. Further, for a given j, all $l_{j,k}$ components implement the same function, f_j. The task of the CoSynthesis Tool is to bind components from the library to nodes within the system such that the functions (f_j) of a bound component ($l_{j,k}$) match those of the node in the system, and the aggregate system performance attributes satisfy the system-level constraints levied by the designer. The data model and flexible retrieval mechanism are further described in Section 3.

2.2.2 System Definition

The input to the HW/SW CoSynthesis tool, S_{in}, is defined as a triple (G, C, B), where:

- G is a dataflow hypergraph, denoted (V, E) where
  V is the set of all nodes, v_i, of the graph G,
  E is the set of all edges, denoted as $\{(v_i, K)\}$, where K is a subset of V.
- C is a set of performance constraints, c_i, that specify S's performance constraints. (Sample c_i are area, weight, power consumption, and time delay.)
- B is a binding set in which a binding, denoted (v_i, $l_{j,k}$), associates one $v_i \in V$ to one and only one $i_{j,k} \in L$. Initially, B can be either the empty set or a user-specified set of bindings.

Output from the HW/SW CoSynthesis tool, S_{out}, is defined similarly to S_{in}. The output system is a triple, (G, A, B), where G and B are defined as above and

- A is a set of system performance attributes. Each $a_i \in A$ is calculated by a specific constraint analyzer in the SCOREBOARD tool and is based either on the performance attributes, p_i, associated with components of the binding set, B, and their satisfaction of the set of performance constraints, $C \in S_{in}$.

Associated with the constraints of the input system, C, and the attributes of the output system, A, is a constraint satisfaction function $X(c_i, a_j)$. This function determines whether or not the attribute a_i of the output system achieves the desired goal set by the input c_i. An

example is area; $X(c_i, a_i)$ compares the output system's area (a simple sum of the area of the bound components) with the designer's input area constraint. The goal of HW/SW CoSynthesis is then

Given: $S_{in} = (G, C_s, B)$, where B is initially either the empty set or a user-specified set of bindings.

Create: S_{out} in which

$\forall$ i, $v_i \in V$, $\exists$ a binding $(v_i, l_{j,k})$ of v_i to a specific $l_{j,k} \in L$ such that

$\forall$ i, $c_i \in$ of C, and $a_i \in A$, the constraint satisfaction function $X(c_i, a_i)$ is satisfied.

2.2.3 Algorithm

Our approach improves on the iterative partitioning technique by incorporating a three step evaluation process for selecting the "best node" to move based on user supplied constraints. Prior to algorithm execution, the nodes of the system are initially bound to an implementation (hardware or software) from the component library. All nodes in the graph are unlocked. The algorithm, outlined in Figure 2, proceeds as follows. Each constraint maintains a separate rank ordered list. During the first step, denoted by [1] in Figure 2, system nodes are inserted into each constraint's ordered list based on the impact of the node's potential movement (rebinding) on the overall system. From the context of the node's score in these ordered lists, constraint ranks are assigned to the nodes during step [2]; these constraint ranks are the scalar values for the node's rebinding vector. Finally, in step [3], the rebinding vectors for the nodes are examined and the node with the shortest vector (Euclidean norm) is selected for rebinding. The node is bound to the alternate implementation and locked, and the three steps are repeated until no further node rebindings are possible.

```
      While (ULTasks ≠ ϕ) {
        FOR EACH(CA) {
[1]       CA->Score(ULTasks);
[2]       CA->Rank(ULTasks);     }
[3]     Task2Rebind = SVector(ULTasks);
        Rebind( Task2Rebind );
        LTasks = LTasks ∪ Task2Rebind;
        ULTasks = ULTasks - Task2Rebind; )
```

Where ULTasks = Unlocked Tasks
CA = Constraint Analyzer
LTasks = Locked Tasks
Svector = ShortestVector routine

Figure 2. SCOREBOARD Algorithm.

The components under consideration for rebinding are initially retrieved from the database using the constraints as part of a criteria-based search (a query). Traditionally, each VHDL-based tool must contain its own parser and mechanism for searching VHDL design units. Our approach is to use a design database and query language facilities rather than incorporating this functionality in each tool within the COMET environment.

3. The Design Database

Many of the tools in the COMET environment, such as tools for partitioning, synthesis, and performance estimation, as well as in industrial design environments, are VHDL-based. The general goals of our design database are (1) that it should "understand" VHDL, and (2) allow flexible retrieval of components specified in VHDL. We accomplish these goals by defining a conceptual data model that is implemented in our database system **Odyssey** [Ven95] [Ven96a]. VHDL can be used as input or obtained as output from the database, in addition to accessing data through other interfaces. We define a general query language that provides an interactive, stand-alone interface, or can be used by tools to retrieve designs. In this way, we can interface with existing tools and additionally allow greater flexibility for browsing and retrieving components from design libraries. Users of the database gain query and view facilities as well as more flexible storage management than with traditional file-based VHDL environments.

Others have developed specialized databases for VLSI CAD [Sie89][Kim90][Nay91][Wag92], however, our research is the first that we are aware of to use a hardware description language as a database description language. Wagner examines some of the issues in using HDLs for database description [Wag95], but models designs at a coarser granularity. Modeling at a finer level of granularity permits queries on information regarding entity ports that may be of prime interest in the CoSynthesis process. For example, numerical accuracy may be an additional constraint imposed by the system specification; during system CoSynthesis, tradeoffs can be made to achieve a particular system numerical accuracy based on the bus widths of the components used in the system.

Our approach to design data modeling and retrieval is to parse and store VHDL source using our conceptual model. The components can be directly accessed through a query interface, either by designers or tools. The instances can also be restored to VHDL so that legacy tools may access designs placed in the database regardless of their source.

$$\vec{V}_C = 0.0\hat{i} + 1.0\hat{j} \quad |\vec{V}_C| = 1.0$$

$$\vec{V}_R = 0.583\hat{i} + 0.0\hat{j} \quad |\vec{V}_R| = 0.583$$

$$\vec{V}_S = 1.0\hat{i} + 0.866\hat{j} \quad |\vec{V}_S| = 1.322$$

For this example, the reverser has the smallest rebinding vector and is the best candidate to rebind for this iteration of the algorithm. It is rebound and locked (eliminating it from consideration in the future). Finally, new system attribute values are calculated (Figure 8) and the algorithm repeats until all nodes have been rebound.

		Cost	Area
Splitter	HW	1	1
Reverse	SW	2	10
Compare	HW	10	10
		23	16

Figure 8. System Attributes after Rebinding.

If the system constraints have not been met, the best solution achieved by the algorithm can be used as the initial bindings and the algorithm re-executed.

5. Results and Analysis

An object-oriented experimental SCOREBOARD system has been prototyped using C++ that accepts a VHDL entity/architecture pair and a constraint description. The VHDL input describes the system as a netlist of instantiated components while the constraint description indicates which constraint analyzers and goals to include in the SCOREBOARD algorithm. Although instantiated components are a subset of the possible VHDL language constructs that can be used to model systems, our approach is extensible to allow us to model any concurrent VHDL task (processes, blocks, concurrent signal assignments, procedure calls, etc.). Currently six primitive constraints are supported: cutset minimization, cutset maximum value, area minimization, area maximum value, cost minimization, and cost maximum value. The "minimization" constraint analyzers attempt to minimize their particular system attribute; the "maximum value" analyzers attempt to minimize a system attribute until a maximum possible value is achieved. Inheritance from a common constraint analyzer base class facilitates the creation and manipulation of additional analyzers within the SCOREBOARD system. The output is a revised VHDL architecture dividing the system into hardware and software components and a VHDL configuration body binding the instantiated components to library elements. Experimental data has shown this algorithm produces better two-constraint designs than existing iterative improvement methods. Further the algorithm's complexity is similar to existing hardware partitioning techniques [She94], namely $O(n^2)$, where n is the number of nodes in the system.

The following two examples depict the attributes of a synthesized system as the SCOREBOARD algorithm iterates to completion. Each example was generated from the same input system, an ISCAS 85 benchmark [ISC85], consisting of 1350 nodes. In the first example, the SCOREBOARD algorithm had three goals: minimize the system cutset, minimize the system area, and balance the respective sizes of the HW and SW partitions. In practice, the third goal is of little value in a HW/SW CoSynthesis environment. It is included here to depict a 3-constraint example and as a further indication of the capability of the algorithm over other partitioning methods. The first two constraints, cutset and area minimization, are plotted in Figure 9. The x-axis shows a history of the iterative rebindings for cutset and area. Each step along the x-axis is one iteration of the algorithm. If the constraints of interest are cutset and area, then the optimal point is approximately around 700. Figure 10 shows the history of rebindings with respect to area balance between hardware and software as well as total area. Although this consideration is artificial in CoSynthesis, it does demonstrate how a third constraint can easily be accommodated in our approach. The balance constraint, as a percentage of each partition's contribution to the whole, is in Figure 10.

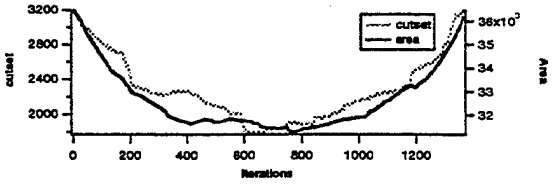


Figure 9. SCOREBOARD Cutset and Area.

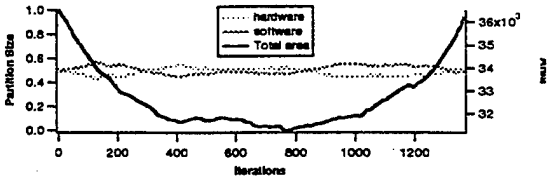


Figure 10. SCOREBOARD Area Balance.

In the second example, a fourth constraint, cost minimization, was added to the analysis of the same

system to illustrate the algorithm's scalability. This constraint adds another dimension to the rebinding vector. Results are presented in Figure 11, Figure 12, and Figure 13. It is apparent by examining minimum values achieved for cutset and area balance in example 2 that a rebinding that was appropriate in the first example is no longer suitable in the second when the additional constraint is considered.

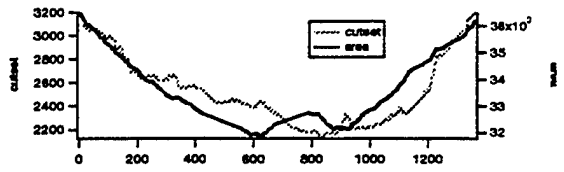


Figure 11. SCOREBOARD Cutset and Area.

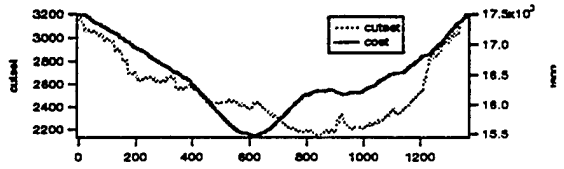


Figure 12. SCOREBOARD Cost and Cutset.

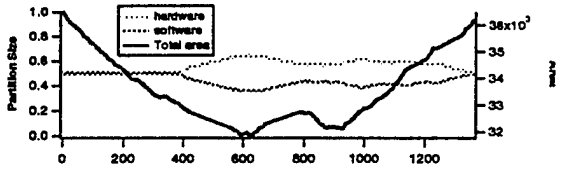


Figure 13. SCOREBOARD Area Balance.

6. Conclusions and Future Work

Conclusions and issues for future work are discussed below.

6.1 Conclusions

The CoSynthesis Tool analyzes candidate solutions and determines the best assignment of resources to hardware and software using an iterative binding algorithm. Our algorithm maintains separate, rank-ordered constraint lists of system nodes that may be rebound for each constraint in the system specification. Our CoSynthesis tool improves on hardware partitioning techniques by selecting the best node for rebinding based on its rebinding vector rather than the first node that is acceptable and allowing additional constraints to be added easily to the evaluation process.

We have proposed and implemented a data model that stores designs described in VHDL and interfaces with legacy tools (VHDL as file input/output) and new state-of-the-art EDA tools (e.g., CoDesign and CoSynthesis tools) to allow design space exploration via criteria-based searching. The contribution is that tools do not have to be scanners, parsers, and query evaluators; designers and tools can continue to work with a widely-used modeling language, and reap the benefits of flexible retrieval.

6.2 Future Work

Future research will cover a broad range of both SCOREBOARD and database refinements. Near-term efforts will formally define and characterize the SCOREBOARD algorithm and an analysis of the quality of the synthesized design. This includes the evaluation of more realistic constraint analyzers and their impact both on the design process and the algorithm. Allowing user-defined constraint weighting to the scalar values of the rebinding vector is an interesting capability. Additionally, the output format will be refined such that the output will include a revised VHDL architecture containing instantiated components representing the hardware and software partitions. The software partitions would be represented as instantiated CPUs and memory executing the software.

Further research could address the granularity of HW/SW CoSynthesis by treating sequential statements of VHDL processes as individual nodes. Designs that define a system's functionality at a more abstract, algorithmic level are not supported in the current version of the algorithm's implementation. Finally, scheduling and resource sharing would greatly aid the HW/SW CoSynthesis effort in that duplicate tasks would not be replicated in the system design.

Areas for future database research include investigation of query optimization and data integration. Data sharing is facilitated since different producers/consumers of design data can use the common database. Data exchange and integration can also be facilitated for other EDA data formats and languages. We have investigated interchange issues for VHDL and the CAD Framework Initiative Design Representation model [Ven96b]. Formats such as SDF [SDF95] for timing delay information pose additional challenges in this area [Dav96].

7. References.

[Ben93] T. Benner, R. Ernst, and J. Henkel. "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Design and Test*, Vol. 10, No. 4, December 1993.

[Bha94] D. Bhatia, Physical Design Automation Course Notes. University of Cincinnati, 1994.

[Car96] C. Carreras, J. Lopez, M. Lopez, C. Delgado-Kloos, N. Martinez, and L. Sanchez. "A Co-Design Methodology Based on Formal Specification and High-level Estimation," *Fourth International Workshop on Hardware/Software CoDesign*, p.28.

[Cha94] P. K. Chan, M. Schalg, and J. Y. Zien. "Spectral K-Way Ratio-Cut Partitioning and Clustering," *IEEE Trans. On Computer-Aided Design*, Vol. 13, No. 9, September 1994, pp. 1088-1095.

[Con94] J. Cong, Z. Li, and R. Bagrodia. "Acyclic Multi-Way Partitioning of Boolean Networks," *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pp. 670-675.

[Dav96] K.C. Davis, S. Venkatesan, and L.M.L. Delcambre, "Sharing Electronic Design Data Via Semantic Spaces," submitted, 1996.

[Gaj94] D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*, Prentice-Hall, Inc, Englewood Cliffs, NJ, 1994.

[Gup93] R. K. Gupta, "Co-Synthesis of Hardware/Software for Digital Embedded Systems," PhD Dissertation, Stanford University, 1993.

[Hen96] J. Henkel and R. Ernst. "The Interplay of Run-Time Estimation and Granularity in HW/SW Partitioning," *Fourth International Workshop on Hardware/Softare CoDesign*, p.52.

[ISC85] Inter. Society on Circuits and Systems, 1985.

[Keu94] K. Keutzer, "Hardware-Software Co-Design and ESDA," *Proc. of 31st Design Automation Conference*, pp. 435-436, 1994.

[Kim90] W. Kim, J. Banerjee, H.-T. Chou, and J.F. Garza, "Object-oriented Database Support for CAD," *Computer Aided Design*, Vol. 22, No. 8, October 1990, pp. 469-479.

[Mar96] T. Marlowe, A. Stoyenko, P. Laplante, R. Daita, C. Amaro, C. Nguyen, and S. Howelll, "Multiple-Goal Objective Functions for Optimization of Task Assignment in Complex Computer Systems," *Control Engineering Practice*, Vol. 4 No. 2, 1996, pp. 251-256.

[Mil95] R. Miller and H. Carter, "Hardware/Software Partitioning in COMET," *Proceedings of the COMET Project Review Meeting*, presentation slides, 1995.

[Nay91] T.K. Nayak, A.K. Majumdar, A. Basu, and S. Sarkar, "VLODS: A VLSI Object Oriented Database System," *Information Systems*, Vol. 16, No. 1, 1991, pp. 73-96.

[SDF95] *Standard Delay Format Specification*, Version 3.0, Open Verilog International, Los Gatos, CA 95032, May 1995.

[She94] N. Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, Norwell, Mass, Second Printing 1994.

[Sie89] E. Siepmann and G. Zimmermann, "Object-Oriented Datamodel for the VLSI Design System PLAYOUT," *Proc. of the 26th ACM/IEEE Design Automation Conference*, Las Vegas, NV, 1989, pp. 814-817.

[Vem94] R. Vemuri, H. Carter, and P. Alexander, "Board and MCM Level Synthesis for Embedded Systems in the COMET Cosynthesis Environment," *Proceedings of the First Annual RASSP Conference*, Arlington, VA, August 1994, pp. 124-133.

[Ven94] S. Venkatesan and K.C. Davis, "A Data Model for VHDL Databases," *VHDL International Users Forum Spring-94*, Oakland, CA, May 1994, IEEE Computer Society Press, pp. 173-182.

[Ven95] S. Venkatesan and K.C. Davis, "Odyssey: An Electronic Design Automation Database," *Proc. of the 2nd International Conference on Applications of Databases*, Santa Clara, CA, December 1995, pp. 147-157.

[Ven96a] S. Venkatesan, "Database Modeling for Electronic Design Automation Environments," Ph.D. Dissertation, Electrical and Computer Engineering and Computer Science Department, University of Cincinnati, Cincinnati, OH 45221-0030, January, 1996.

[Ven96b] S. Venkatesan and K.C. Davis, "A Meta-model and Semantic Mapping Methodology for Hardware Design Data Management," *Journal of Integrated Computer-Aided Engineering*, Vol. 3, No. 1, January 1996.

[Ven96c] S. Venkatesan and K.C. Davis, "Flexible Component Retrieval for Co-Design," submitted, 1996.

[Wag92] F.R. Wagner, L.G. Golendziner, J. Lacombe, and A. H. Viegas de Lima, "Design Version Management in the STAR Framework," *IFIP92*, edited by M. Newman and T. Rhyne, Elsevier Science Publishers B.V. (North-Holland), March 1992.

[Wag95] F.R. Wagner, "Design Management Requirements for Hardware Description Languages," *Proceedings EURO VHDL 95*, 1995.

[Wei91] Y.C. Wei and C.K. Cheng, "Ratio Cut Partitioning for Hierarchical Designs," *Transactions on Computer-Aided Design*, Vol. 10, No. 7, July 1991, pp. 911-921.

[Yeh95] C. Yeh, C. Cheng, and T. Lin, "Optimization by Iterative Improvement: An Experimental Evaluation on Two-Way Partitioning", *IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems*, Vol. 14, No. 2, February 1995, pp. 145-153.

APPENDIX F:

A Retiming Based Relaxation Heuristic for Resource-Constrained Loop Pipelining *

Vinoo Srinivasan and *Ranga Vemuri*†

Laboratory for Digital Design Environments
Department of ECECS
P.O. Box 210030
University of Cincinnati
Cincinnati, OH 45221-0030

*This work was partially supported by the ARPA RASSP program and monitored by the Wright Lab, US-AF under contract number F33615-93-C-1316 and ARPA HPCC program monitored by the FBI under contract number J-FBI-93-116

†Author for Correspondence, (513)-556-4784 (Voice), (513)-556-7326 (FAX), *Ranga.Vemuri@UC.EDU*

A Retiming Based Relaxation Heuristic for Resource-Constrained Loop Pipelining.

Abstract

This paper presents a fast and efficient heuristic for pipelining a loop under resource-constraints. The loop is represented as a dependence graph, G, whose nodes are operations that are bound to available resources and edges denote the data dependencies between the operations. The data dependencies restrict the degree of parallelism that can be achieved while scheduling the graph. We propose a fast retiming based graph transformation technique which relaxes the data dependencies in the graph while maintaining functional equivalence. Relaxing data dependencies provides more flexibility for the scheduler to schedule operations, thereby leading to faster throughput. Our objective is to obtain a retimed graph which when scheduled achieves an optimal/near-optimal pipelined steady state throughput. A detailed algorithm is presented to solve the problem. We provide results that illustrate the effectiveness of our algorithm.

2 Definitions and Terminology

2.1 Specification Model

The target architecture model for our retiming framework consists of a resource set $\mathcal{R} = \{R_1, \cdots, R_n\}$. For a given operation n, $t(n)_R$ is the execution time for the operation n on the resource $R \in \mathcal{R}$. If the operation n cannot be executed on the resource R then $t(n)_R = \infty$. The loop body is represented by a *Dependency Graph*. We assume that the graph is executed several times corresponding to the iterative computations of the loop, involving varying data sets over time. Our loop representation is an extension of that in UNRET [9] and is similar to the signal processing data flow graph representation in [19].

Definition 2.1 A *dependence graph* (**DG**) is a directed graph denoted by a 5-tuple, $\mathcal{DG} = (V, E, \lambda, \delta, \beta)$. V is the set of nodes representing the operations in the loop. E is the set of directed edges corresponding to the dependencies. $\lambda : V \mapsto \mathcal{N}$ is a mapping which assigns an *iteration index* to each node in the DG. $\delta : E \mapsto \mathcal{N}$ is a mapping which assigns an non-negative integer *delay* value to all the edges in the DG. $\beta : V \mapsto \mathcal{R}$ is a binding of each node to a resource. □

Iteration index (λ): Since the *DG* represents an iterative algorithm, each iteration of the DG execution invokes all the operations in the graph once. Thus if the DG is executed over N iterations, then each node $v \in V$ has N instances $v_1, v_2 \cdots v_{N-1}, v_N$ where v_i is that instance of the node v corresponding to the ith iteration of the DG. The subscript i in v_i is the iteration index (λ).

Dependency delay (δ): Edges in the DG represent data dependencies. A delay of nD on an edge is equivalent to having n delay units on that edge. An edge $u_0 \xrightarrow{k} v_0$ (an edge from node u_0 to node v_0 with a k delay units) implies data dependence from instance u_c to instance v_{c+k}, for $c \geq 0$. In general an edge $u_i \xrightarrow{k} v_j$ implies data dependence from instance u_{i+c} to v_{j+k+c}, for $c \geq 0$.

Depending on the level of granularity, nodes in the *DG* can range from simple operations like multiplications and additions to complex macro operations like fast Fourier transforms and matrix multiplications. Correspondingly, the resources can range from simple multipliers and adders to off-the-shelf microprocessors and FPGAs. In the rest of this paper we will use the word task synonymous with nodes and operations. A *DG* is considered *legal* only if the following three conditions are satisfied:

- $\forall u \in V : \lambda(u) \geq 0$ (1)
- $\forall\ u \rightarrow v\ \in\ E : \delta(u \rightarrow v) \geq 0$ (2)
- $\forall$ cycles $c \in G : \delta(c) > 0$, where $\delta(c) = \sum_{u \rightarrow v\ \in\ c} \delta(u \rightarrow v)$ (3)

The condition (1) does not permit nodes with negative iteration indices, (2) forces all edges to have non-negative delays and (3) eliminates the existence of any cycle with zero or negative delay. An *Initial DG* is a *DG* such that $\forall v \in V : \lambda(v) = 0$. Dependencies between task instances belonging to the same steady state execution are called *local dependencies* while those between task instances of different steady states are called *global dependencies*. All edges with $\delta(e) = 0$ are called *local edges* and denote local dependencies, while edges with $\delta(e) > 0$ are called *global edges* and denote global dependencies.

2.2 Scheduling a DG

Given a dependence graph $G = (V, E, \lambda, \delta, \beta)$, we define the set $E_l = \{e \in E : \delta(e) = 0\}$ to be the set of local dependencies. Only the local dependencies affect the schedulability of G. A node $v \in V$ is a *head node* if and only if, there exists no node u such that $u \rightarrow v \in E_l$. A node $v \in V$ is a *tail node* if and only if, there exists no node w such that $v \rightarrow w \in E_l$. For any node $u \in V$, the execution time for that node when executed on the resource to which it is bound ($\beta(u)$) is called *latency* of that node, $l(u)$. For any path $u \rightsquigarrow v \in E_l$ (path involving only the edges in E_l), the *latency* of the path, $l(u \rightsquigarrow v)$, is equal to the sum of latencies of the nodes that belong to the path. Mathematically,

$$\forall\, u \in V \;:\; l(u) = t(u)_{\beta(u)}$$
$$\forall\, u \rightsquigarrow v \;\in\; E_l \;:\; l(u \rightsquigarrow v) \;=\; \sum_{n \in (u \rightsquigarrow v)} l(n) \tag{4}$$

A path, $p \in E_l$, is a *critical path* if for all paths $p' \in E_l : l(p) \geq l(p')$. CPL (Critical Path Latency) denotes the latency of the critical path in the dependence graph.

Definition 2.2 : $\mathcal{S}(G)$ (Schedule of G)
A schedule of a the graph $G = (V, E, \lambda, \delta, \beta)$ is a mapping $S : V \mapsto \mathcal{N}$ such that:

$$\forall (u \rightarrow v) \in E_l :\; S(v) \geq S(u) + l(u)$$

□

The schedule of a graph, $\mathcal{S}(G)$ - definition 2.2, is an assignment of start times for the execution of all the tasks in the graph, such that the local data dependencies are not violated. The Initiation Interval (II) of a loop is the time interval between consecutive executions of its steady state. Given a schedule of a loop, the initiation interval of the loop for that schedule, II_S, is the difference between the time at which all scheduled tasks finished execution and the earliest time at which any task was scheduled.

$$II_S \leftarrow \max_{u \in V} (S(u) + l(u)) - \min_{u \in V} S(u) \tag{5}$$

Since all the tasks in the *critical path* have to be scheduled, for any schedule $\mathcal{S}(G)$, $II_S \geq CPL$.

2.3 Theoretical bounds on Initiation Interval

It is clear that CPL poses a bound on the the II of the graph. The resource constraints and the recurrences present in the DG also restrict the II of the steady state [19, 12, 9]. Consider a DG with k multiplication operations, and a resource set with n multipliers, then, assuming multiplication takes unit time, it will take at least $\lceil k/n \rceil$ time units to schedule all the multiplication operations. The maximum of such time bounds over all resource types is the Minimum Initiation Interval (MII) due to resource constraint, represented as MII_{res}. In the presence of a recurrence r in the DG, the steady state execution time is lower bounded by $\lceil l(r)/\delta(r) \rceil$ time units to assure proper execution of the recurrence. The maximum of such bounds over all recurrences in the graph is the MII due to recurrences, represented as MII_{rec}. Mathematically,

$$MII_{res} = \max_{R_i \in \mathcal{R}} \frac{\sum_{\forall u \,:\, \beta(u) = R_i} l(u)}{n_i} \quad ; \quad MII_{rec} = \begin{cases} 0 & \text{if there is no recurrence} \\ \max\limits_{r \in G} \dfrac{l(r)}{\delta(r)} & \text{otherwise} \end{cases}$$

where R_i is a specific resource type from the resource set $\mathcal{R}$ and n_i is number of such resources available. r is a recurrence in the DG. $l(r)$ is the sum of the latencies of all the nodes in r and $\delta(r)$ is sum of the delays of all the edges in r.

Definition 2.3 The **Minimum Initiation Interval** (MII_G) achievable by any schedule for a given graph is the maximum of the two lower bounds discussed above.

$$II_S \geq MII_G = max(MII_{res}, MII_{rec})$$ □

2.4 Retiming the Dependence Graph

Definition 2.4 : $r(G)$ (Retiming of a dependence graph G)
The retiming operation transforms the graph $G = (V, E, \lambda, \delta, \beta)$ into a new graph $G_r = (V, E, \lambda_r, \delta_r, \beta)$, such that:

$$\forall (u \rightarrow v) \in E : \delta_r(u \rightarrow v) - \delta(u \rightarrow v) = (\lambda_r(u) - \lambda(u)) - (\lambda_r(v) - \lambda(v))$$ □

A retiming operation is *legal* if it always transforms a legal dependence graph G to a retimed graph G_r which is also legal. Recollect that a legal DG is one which satisfies conditions (1), (2), and (3). In the rest of our discussion we restrict ourselves to *legal* retiming operations. If G_r is a retimed graph of G derived by a legal retime operation, then G_r is *functionally equivalent* to G [20]. The retiming operation does not change the MII of a graph. However, retiming may introduce delays on *local edges* thereby eliminating *local dependencies*. Eliminating local edges that belong to the critical path may reduce the CPL, which might lead to faster schedules.

Figure 1 shows an example of how retiming is used to generate pipelined schedules with better throughput. The DG has four tasks A, B, C and D and two resources $R1$ and $R2$. Tasks A, C are bound to $R1$ and tasks B, D are bound to $R2$. For simplicity we assume that all four tasks take unit time to execute. We see that the schedule for the original graph tasks 3 cycles per iteration of the loop ($II = 3$) while the retimed graph has an II of 2 cycles for the steady state. Also notice that after retiming we achieve a pipelined schedule (Figure 1-b) while the schedule produced for the initial graph is non-pipelined (Figure 1-a).

3 Resource-Constrained Loop Pipelining

In this section we present our algorithm that attempts to generate an optimal resource-constrained pipelined schedule for a given dependence graph representing a loop. We consider a pipelined schedule optimal if the steady state initiation interval of the schedule (II_S) is equal to the minimum initiation interval of the loop (MII_G) as given in definition 2.3. Given the initial graph of the loop we try to produce the retimed graph which when scheduled achieves the best possible steady state throughput.

Since we want to achieve the best throughput, the aim of the retiming algorithm must be to eliminate as many local dependencies as possible. Figure 2 shows two examples where retiming is used to eliminate local dependencies in a DG. The underlying retiming operation used in Figure 2 is the one referred to

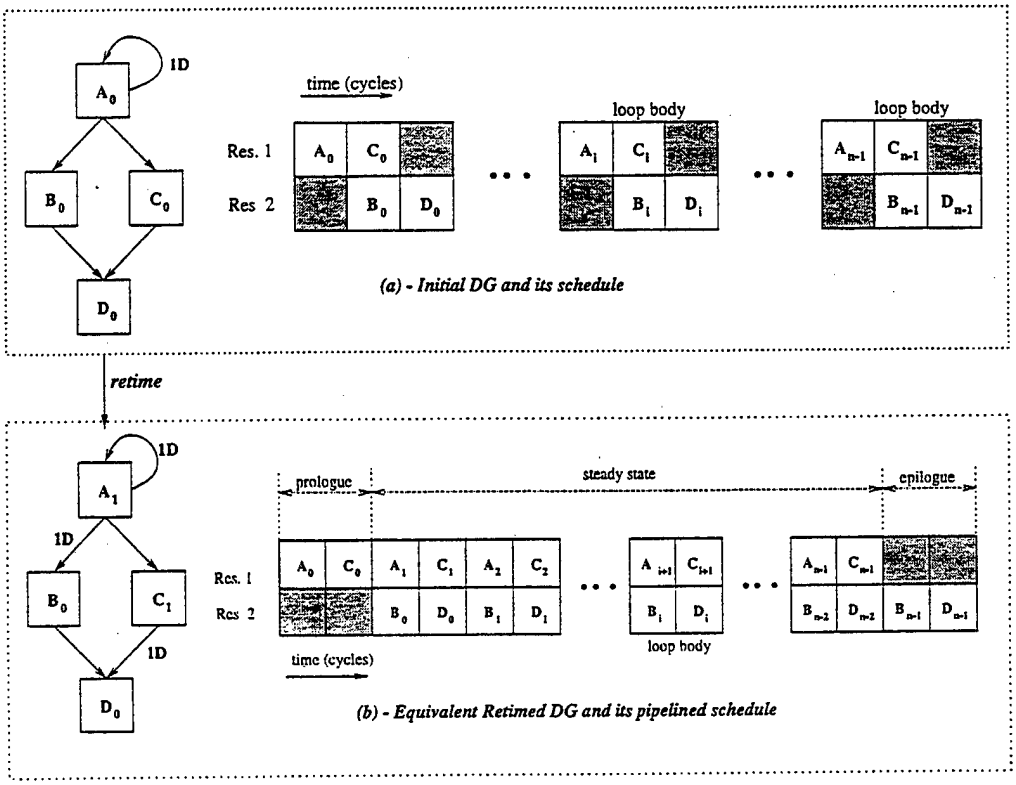


Figure 1: **An Example of Retiming to generate pipelined schedules**

as *nodal transfer* in [12], which is same as dependence retiming transformation presented in [9]. We shall call it the the function *shift_node*. For a given node $v \in V$, and for a positive integer k the function $shift_node(v, k)$ performs the following steps:

- $\lambda_r(v) \leftarrow \lambda(v) + k$
- $\forall (u \rightarrow v) \in E : \delta_r(u \rightarrow v) \leftarrow \delta(u \rightarrow v) - k$
- $\forall (v \rightarrow w) \in E : \delta_r(v \rightarrow w) \leftarrow \delta(v \rightarrow w) + k$

The $shift_node(v, k)$ function transforms a given DG into an equivalent retimed DG satisfying definition 2.4. However, $shift_node(v, k)$ will be a *legal* retiming operation only if for all edges $u \rightarrow v \in E : \delta(u \rightarrow v) \geq k$, otherwise edges with negative delays will be created. A DG is defined to be *systolic* if it has no local dependencies [21], i.e. $\forall e \in E : \delta(e) > 0$. For a systolic DG it is trivially possible to obtain a schedule

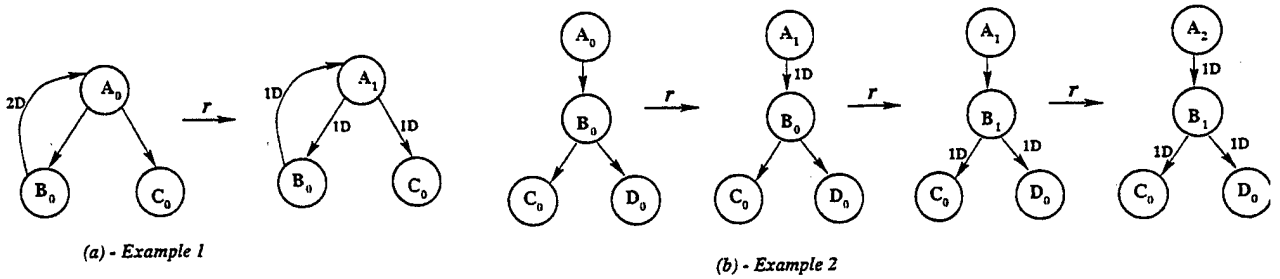


Figure 2: **Retiming Transformations to Eliminate Local Dependencies**

Algorithm 3.1 (Retiming an Acyclic *DG*)
$G = (V, E, \lambda, \delta, \beta)$: *The Initial DG to be retimed.* ▷ The initial graph is acyclic
procedure *retime_acyclic_DG*(G)
begin
  while ($\exists$ *head* node $u \in V$ such that ($\exists\ u \rightarrow v \in E : \delta(u \rightarrow v) = 0$)) **do**
    shift_node($u, 1$);
  return G
end

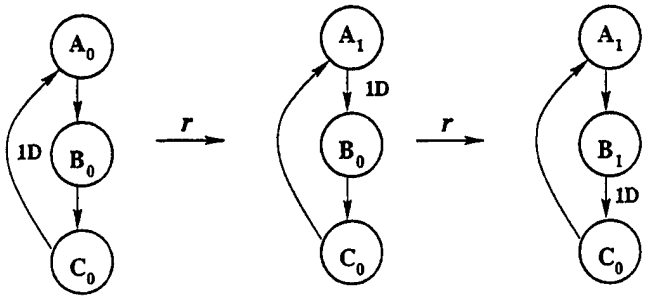


Figure 3: **Retiming a cyclic *DG***

that is optimal with II_S equal to MII_{res}.

If the initial graph has no cycles then it is always possible to introduce positive delays on all its edges and achieve the optimal throughput. Algorithm 3.1 is simple procedure which eliminates all local edges in an acyclic *DG*, just by making calls to the *shift_node*() function. Figure 2-(b) illustrates the flow of this algorithm when applied to a acyclic *DG*. In Algorithm 3.1, since the node u is a head node, we do not create any edges with negative delays. In the case of *DG*s with cycles, it is not always possible to eliminate all local edges. Consider the initial cyclic *DG* in Figure 3. Any legal retimed graph of the initial graph always has two local data dependencies. In more general terms it can be easily proved that for all cycles c in the graph, $\delta(c)$ (sum of the delays of the edges in the cycle) does not change with retiming. Thus for *DG*s with cyclic dependencies, there are cases when we can only shift around delays (i.e. reducing the delay value on certain edges and adding it to others) rather than creating new delays.

Although for any cycle, c, in the *DG*, $\delta(c)$ is constant over retiming, the number of positive delay edges in

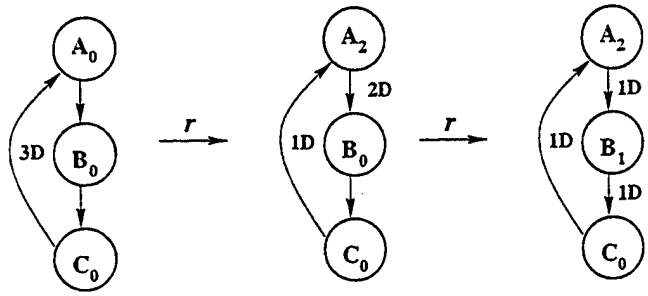


Figure 4: **Relaxing a cyclic *DG***

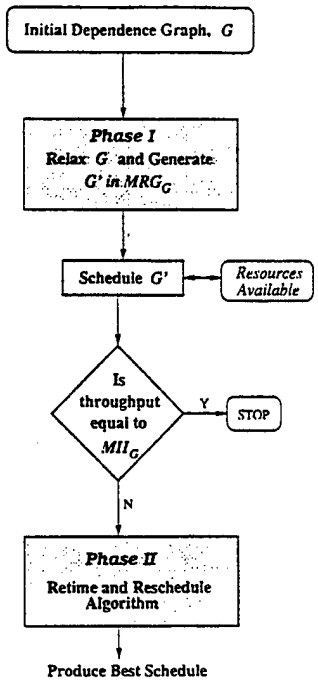


Figure 5: **Resource-constrained Loop Pipelining Methodology**

obtain the pipelined throughput. If the throughput achieved by the scheduler is equal to MII_G, then the optimal steady state throughput has been achieved and we do not proceed to phase two. We use a simple list-scheduler [22] with mobility of the nodes as the primary priority. In the second phase we pass G', the output of phase one, to a retime and schedule algorithm. We now present the details of both the phases of our algorithm.

3.1.1 Phase I Algorithm

In the first phase we try to transform the given initial graph into an MRG. Our approach is presented in Algorithm 3.2. Before invoking the algorithm we identify the set of edges in the DG which belong to recurrences. A directed edge from node u to node v belongs to the recurrence set, $\mathcal{R}$, if there exists a directed path from v to u (i.e. there is a cycle involving the edge $u \rightarrow v$). Mathematically, $\mathcal{R} = \{u \rightarrow v \in E \mid \exists\ path\ v \rightsquigarrow u\}$. Edges that belong to $\mathcal{R}$ are called *recurrence* edges and the rest are called *non-recurrence* edges. The procedure $relax_DG()$ in Algorithm 3.2 has two while loops. The first while loop transforms all non-recurrence local edges into global edges. The second while loop tries to introduce delays on local edges that belong to $\mathcal{R}$.

Relaxing non-recurrence edges: This is done in the first *while* loop of the algorithm 3.2. Consider an zero delay edge $u \rightarrow v$ that does no belong to any recurrence. We follow a simple approach to introduce a unit delay on this edge without decrementing existing delays on any other edge of the graph. For all nodes, n, belonging to the set that includes the node u and all nodes from which u can be reached, perform $node_shift(n, 1)$. The above retiming will introduce an additional unit delay on all out edges from u

Algorithm 3.2 (Phase I - Relaxing a *DG*)

$G = (V, E, \lambda, \delta, \beta)$: *The Initial DG to be relaxed until it is a MRG.*
$\mathcal{R}$: *The set of recurrence edges that belong to E*

```
procedure relax_DG(G)
begin
  ▷ Eliminate all non-recurrence local edges
  while (∃(u → v) ∈ E s.t. (δ(u → v) = 0) ∧ (u → v ∉ R)) do
  begin
    for each n ∈ ({u} ∪ {k | ∃ a path k ⇝ u}) do
      shift_node(n, 1)
  end while
  ▷ Now try to eliminate local edges belonging to recurrences.
  while ( [u → v, shift] ← get_next_relaxable_edge(G) ) do   ▷ loops until function returns NULL
  begin
    shift_node(u, shift)
    for each edge t → u ∉ R do
    begin
      if (δ(t → u) < 1) then
        d ← 1 − δ(t → u)
        for each n ∈ ({t} ∪ {k | ∃ a path k ⇝ t}) do
          shift_node(n, d)
      end if
    end for
  end while
end
```

(excluding the self loop), while not introducing any new local dependency in the graph. Thus the edge $u \rightarrow v$ is no longer a local edge.

Figure 6 illustrates non-recurrence edge relaxing. (a) is the initial graph. The non-recurrence zero delay edge $B \rightarrow D$ is selected to be relaxed. A, C are the nodes from which B can be reached. Hence, the $shift_node(n, 1)$ function is performed on nodes B, A and C. (b) is the graph obtained after all $shift_nodes$ are performed. Notice that for all edges from one of the three nodes (A, B, C) to any of the remaining nodes, the delay on the edge is increased by one unit. So in (b) we see that delays are introduced on the edges $B \rightarrow D$ and $C \rightarrow D$. We continue this procedure until all local dependencies are eliminated. Figure 6-(c) shows the graph obtained after all the local dependencies are eliminated.

Relaxing recurrence edges: This is done in the second *while* loop of the algorithm 3.2. Consider a zero delay recurrence edge $u \rightarrow v$ in the graph. The approach taken for non-recurrence edge will not work here because u is reachable from itself. However if all recurrence edges incident on u (excluding self loop) have delay $> kD$ units, for some positive integer k, then we can perform $shift_node(u, k-1)$. This will introduce a positive delay on the recurrence edge $u \rightarrow v$ and all recurrence edges that are incident on u will remain positive. However non-recurrence edges incident on u may be transformed into local dependencies. These new local dependencies can be eliminated through the approach previously discussed. As stated

Algorithm 3.3 (Select an Edge to Relax)

$G = (V, E, \lambda, \delta, \beta)$: *A node has to be selected from then input graph G*
selected[e] : *selected is a global boolean array. selected*[e] *denotes if the edge e has already been selected or not. Initially all edges are marked unselected.*
$\mathcal{R}$: *The set of recurrence edges that belong to E*

function *get_next_relaxable_edge(G)* : $(u \rightarrow v : E,\ shift : int)$
begin
  while ($\exists$ *edge* $(u \rightarrow v) \in \mathcal{R}$ such that (not *selected*$[u \rightarrow v]$) $\wedge$ $(\delta(u \rightarrow v) = 0)$)
    $E' \leftarrow \{(t \rightarrow u) \in \mathcal{R} \mid t \neq u\}$
    $shift \leftarrow \min_{e \in E'} \delta(e)$
    if $(shift > 1)$ **then**
      selected$[u \rightarrow v] \leftarrow 1$
      return $(u \rightarrow v,\ shift - 1)$
    end if
  end while
  return NULL
end

earlier the sum of the delays on any recurrence is constant. So, essentially, we select nodes belonging to recurrences that have excess delays on all recurrence edges incident on them and redistribute the excess delay to their outgoing edges.

The function *get_next_relaxable_edge(G)* selects the candidate recurrence edge to be relaxed next. The function also returns an integer value, *shift*, by which the selected edge can be relaxed. The selection function is shown in algorithm 3.3. This function selects nodes belonging to recurrences such that all recurrence edges (excluding self loops) incident on it have a delay greater than one. If no such unselected node exists then it returns null. The integer value, *shift*, returned by this function is equal to one less than the least delay on the recurrence edges mentioned above. For each edge, $u \rightarrow v$, selected by the selection function, $shift_node(u, shift)$ is performed. Thus delays on all outgoing edges of u will be increased by $shift$ and delays on all edges incident on u will be decreased by $shift$. This will eliminate the local dependency $u \rightarrow v$. Due to way *shift* was computed, positive delays are maintained on all recurrence edges incident on u. The only local dependencies that may be created are on the non-recurrence edges incident on u. However using the technique discussed before to relax non-recurrence local dependencies, these new local edges are eliminated.

Figure 7 shows an example of relaxing recurrence edges. In Figure 7-(a) the local recurrence edge $B \rightarrow C$ is chosen to be relaxed and the value of $shift$ is 2 (the excess delay on the edge $D \rightarrow B$). $shift_node(B, 2)$ is performed to distribute the excess delay to the local node $B \rightarrow C$. Notice that in Figure 7-(b) the edge $D \rightarrow B$ now has a unit delay. In order to maintain the unit delay on the non-recurrence edge $A \rightarrow B$, a $shift_node(A, 2)$ is performed. The graph (b) is an MRG and so the selection function of Algorithm 3.3 returns null.

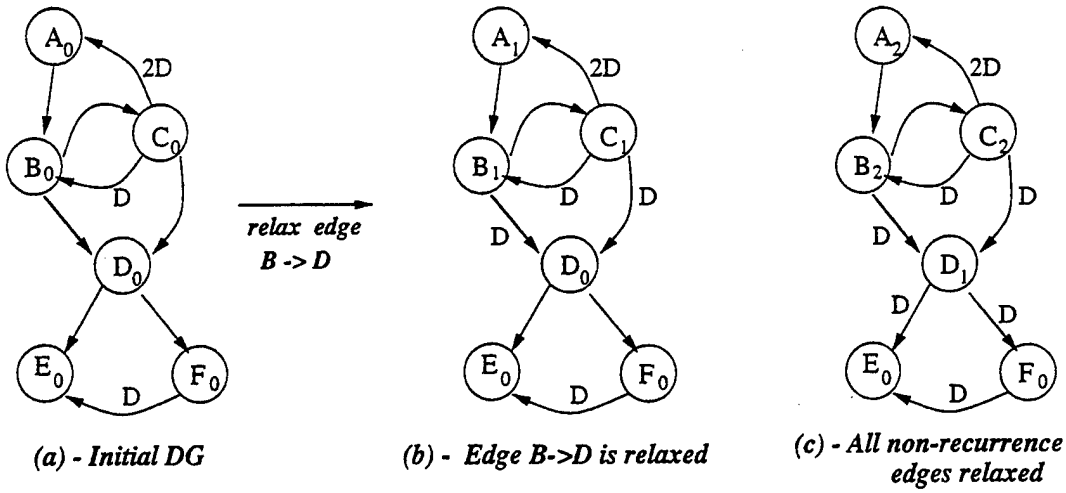


Figure 6: **Relaxing Non-Recurrence Edges**

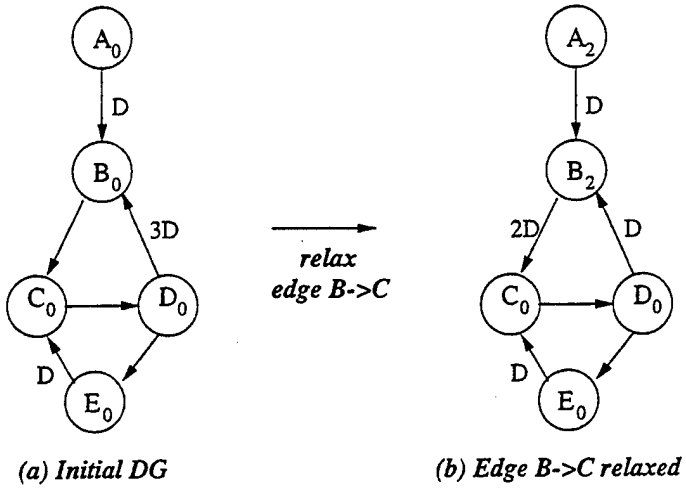


Figure 7: **Relaxing Recurrence Edges**

3.1.2 Phase II Algorithm

The phase two algorithm is invoked if the MRG obtained after phase one does not produce a schedule that achieves the optimal steady state throughput. Since the schedule is not optimal, the resources are not fully utilized. There are gaps in the schedule where certain resources are idle. These gaps are created due to presence of certain local data dependencies. We identify such dependencies and introduce delays on them at the expense of introducing other local dependencies. The retimed graph is scheduled again and the process is continued either until the optimal throughput is achieved or until all edges are tried. The best throughput is reported if the optimal value is not achieved.

Figure 8 illustrates our phase two algorithm. The graph in (a) is the DG obtained after phase one. The graph has four tasks. Tasks A and B are bound to the processing element 2 ($PE2$) and have execution times of 40 and 60 cycles respectively. Tasks C and D are bound to the processing element 1 ($PE1$) and have execution times of 50 and 45 cycles respectively. The MII_{res} is equal to 100 (max(60+40, 50+45)).

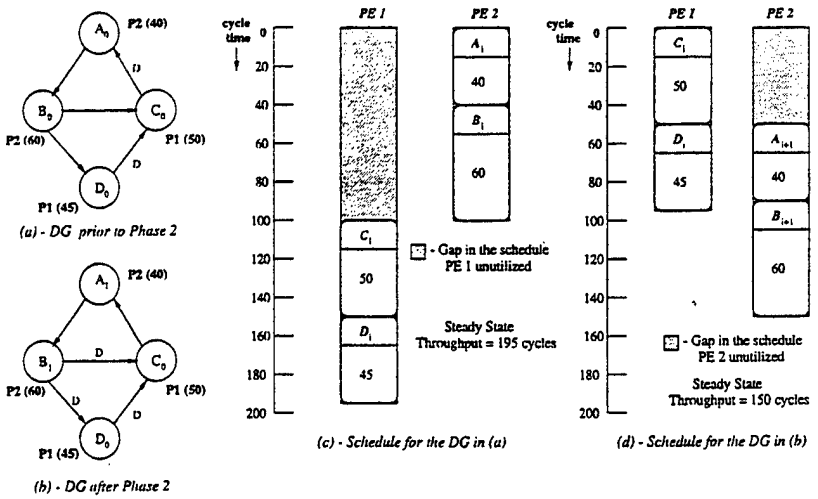


Figure 8: **Illustration of the Phase 2 algorithm**

There are two recurrences in (a) - $A \rightarrow B \rightarrow C \rightarrow A$ and $A \rightarrow B \rightarrow D \rightarrow C \rightarrow A$. MII_{rec} is equal to 150 (max(150/1, 195/2)). Thus the MII_G for the DG is 150 (max(150, 100)). Figure 8-(c) is the schedule obtained for the graph in (a). The throughput obtained for (a) was 195 cycles. We notice that $PE1$ is not utilized for the first 100 cycles of the schedule, which is what we call *gap* in the schedule. The gap is created due to the local dependency $B \rightarrow C$. The task C has to wait until task B completes execution. Hence, local edge $B \rightarrow C$ is chosen to be relaxed. To create a delay on this edge, $shift_node(B, 1)$ is invoked, but since the edge $A \rightarrow B$ is also a local edge, $shift_node(A, 1)$ is in turn called. In general $shift_node()$ is recursively invoked until a legal DG is obtained.

Figure 8-(b) is the DG obtained after the edge $B \rightarrow C$ is relaxed. Notice that the edge $C \rightarrow A$ is now a local edge. The iteration indices of A and B are incremented by one. Figure 8-(d) is the schedule for the retimed DG in (b). This schedule is a pipelined schedule representing the steady state execution of the loop. The schedule achieves the optimal steady state throughput of 150 cycles per execution. If the optimal solution were not achieved, the algorithm would identify the local edges causing gaps and continue the relaxation process. A resource is considered *alive* until the time the last task scheduled on it completes execution. It is a *critical resource* if it is alive beyond the optimal schedule time of the steady state (MII_G). Gaps on non critical resources are ignored. If there are more than one unselected local edges causing gaps, then we choose one of them based on priorities such as criticality of the resource, gap size, and edges belonging to the critical path.

UNRET [9] also uses a retime and reschedule approach. But, instead of looking for gaps in the schedule like our phase 2 approach, it picks an unselected head node from the DG, performs $shift_node$ on it and reschedules the retimed DG. The process continues either until optimal throughput is achieved or until all nodes are selected. The phase 2 approach we follow is efficient because each retiming move is dependent on the feedback from the schedule produced, rather than arbitrarily choosing a head node as in [9]. The main difference between our resource constrained loop-pipelining methodology, presented in Figure 5, and that in [9, 11] is the lack of phase 1 in the later. The advantage of the relaxation scheme followed in phase 1 is that there may be no need to resort to the phase II algorithm because the relaxed graph obtained as

Example Number	Num. of Tasks	Num. of Dependencies	MII_G (cycles)
1	40	80	1282
2	50	150	1852
3	100	300	2788
4	100	500	3281
5	200	1000	6774
6	300	1500	8744
7	400	1600	12008
8	400	2000	13264
9	500	2000	15353
10	500	2500	15467

Table 1: **Design Data for the Test Examples**

the result of phase I produces the optimal schedule. Even in the case when phase II cannot be avoided, the convergence time of phase II when preceded by phase I is usually much faster than just phase II alone because in the former case the second phase starts off with as maximally relaxed graph. In the next section we present results to justify the above claims.

4 Results

In this section we present results of our resource-constrained loop-pipelining methodology shown in Figure 5. We compare our algorithm against the retime and schedule scheme in UNRET [9]. We have implemented all algorithms in C++ on a Sparc 5 Unix workstation running at 143Mhz clock. The reason why we chose UNRET for our comparison is that the later has been compared against several known pipelining schemes and proved effective in [9].

We have implemented a dependence graph generator that can produce synthetic graphs of varying complexities. The generator takes a input the number of nodes, number of edges, number of resources available, execution time range and maximum delay on any edge. Table 1 presents the details of the synthesized dependence graphs generated that are used to study the efficiency of our methodology. To keep the scheduler simple, we consider two resources like the example in Figure 8. Each task is mapped randomly to one of the resources, and the execution time is randomly selected from the uniformly distributed interval $[20; 100]$ cycles. The maximum delay value on any edge is 3 delay units and the probability of an edge being a local dependency is 0.8. All graphs generated are *legal* dependence graphs. Table 1 also shows the theoretical bound on the initiation interval of any pipelined execution for all the ten test graphs.

Table 2 compares our loop-pipelining algorithm against that of UNRET for the 10 examples in Table 1. Column 2 (C2) is the amount of execution time spent on Phase I of the algorithm, C3 is the time spent in Phase II, and C4 is the total execution time. Column 6 is the time taken by retiming approach presented in UNRET. All times are reported in milli seconds. Columns 5 and 7 are the cycle times of the fastest

Example Number	Two Phased Algorithm				UNRET		Speedup (times)
	Phase I Time (ms)	Phase II Time (ms)	Total Time (ms)	II_S (cycles)	Time (ms)	II_S (cycles)	
1	1.1	0	1.1	**1282**	43.8	**1282**	39
2	0.8	7.8	8.6	**1852**	184.3	1881	21
3	3.6	0	3.6	**2788**	215.0	**2788**	60
4	2.3	486.1	488.4	3519	894.5	3607	2
5	5.0	36.0	41.0	**6774**	1240.0	**6774**	30
6	7.9	45.7	53.6	**8744**	1195.1	**8744**	22
7	23.3	87.7	111.0	**12008**	4196.2	**12008**	38
8	13.3	60.1	73.4	**13264**	5236.8	**13264**	71
9	19.0	0	19.0	**15353**	2645.0	**15353**	139
10	21.1	394.0	415.1	**15467**	8123.0	**15467**	20

Table 2: **Resource Constrained Loop Pipelining : Results**

pipelined schedule produced by our approach and UNRET's approach respectively. The numbers in bold indicate that the optimal throughput time was achieved. Our algorithms achieves the optimal throughput for 9 of 10 examples. For example 4 both approaches failed to produce the optimal throughput.

The result we want to highlight in Table 2 is the speed up in the execution times. For the 10 examples, on an average, our approach is about *44 times* faster than that of UNRET. Only for example 4, where both approaches fail to produce the optimal result, we do not see a substantial speed up. UNRET is slow because of the time it spends in the scheduler. Each time a *shift_node* operation is done, the graph is rescheduled. As the size of the graph increases, scheduling becomes much slower. Our phase 2 algorithm also uses a retime and reschedule approach like UNRET, but we differ in the way the graph is retimed. The reason for the speed up is the presence of the *relaxation* algorithm of phase I. For examples 1, 3, and 9 phase II was not needed. For the remaining examples, although phase II was needed, it converged toward the optimal solution much faster than UNRET. Thus, our approach is atleast as efficient UNRET in terms of throughput achieved for a given loop, while at the same time it's execution time is several magnitudes faster than the later.

5 Conclusion

This paper presented an efficient two phased algorithm for resource-constrained loop pipelining. Our algorithm extensively uses *retiming* techniques [7] to generate pipelined schedules. The focus of our algorithm was to achieve the best possible steady state throughput for a given loop while expending minimal computation time. The effectiveness of our algorithm was illustrated through several synthetically generated dependence graphs, representing loops of varying complexities. Results show that execution time of our algorithm is much faster than the scheme in UNRET [9] while not sacrificing the quality of the steady state

throughput. Currently we are applying the our loop-pipeling algorithm to a hardware/software codesign framework to produce pipelined hardware-software codesins.

References

[1] K. Hwang, F.A. Briggs. *Computer Architecture and Parallel Processing.* The MIT press, Cambridge, Massachussets, 1984.

[2] M. Lam. "Software Pipeling: An effective scheduling technique for VLIW machines". In *Proc. of SIGPLAN*, pages 318–328, June 1988.

[3] A. Aiken, A. Nicolau. "Perfect Pipelining: A new loop parallelization technique". In *Lecture notes in Computer Science*, volume 300, pages 221–235, March 1988.

[4] R. Potasman, J. Lis, A. Nicolau, D. Gajski. "Percolation Based Synthesis". In *Proc. ACM/IEEE Design Automation Conference*, pages 444–449, 1990.

[5] A. Aiken, A.Nicolau. "A Realistic Resource-Constrained Software Pipelining Algorithm". In *Advances in Languages and Compilers for Parallel Processing*, pages 85–92, March 1996.

[6] B.R. Rau, C.D. Glaeser. "Some scheduling techniques and an easily scheduable horizontal architecture for high performance scientific computing". In *Proc. of the 14th Annual Workshop on Microprogramming*, pages 183–198, Oct. 1981.

[7] C.E. Leiserson, J.B. Saxe. "Retiming Synchronous Circuitry". In *Algorithmica*, pages 5–35, 6:5-35, 1991.

[8] M.C. Papefthymiou. "Understanding Retiming Through Maximum Average-Delay Cycles.". In *Mathematical systems theory.*, volume 27, 1994.

[9] F. Sánchez, J. Cortadella. "Resource-constrained software pipelining for high-level synthesis of DSP systems". In *Algorithms and Parallel VLSI Architectures III*, pages 377–388, 1995.

[10] M. Potkonjak, J. Rabaey. "Optimizing Resource Utilization Using Transformations". In *IEEE transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 13, March 1994.

[11] Fermín Sánchez. *"Loop Pipelining with Resource and Timing Constraints"*. PhD thesis, UPC. Universitat Politècnica de Catalunya, Spain, October 1995.

[12] V.K. Madisetti. *VLSI Digital Signal Processors: An Introduction to Rapid Prototyping.* IEEE Press, 1995.

[13] S. Huang, J. Rabaey. "Maximizing the throughput of high performance DSP applications using behavioral transformations". In *Proceedings of EDAC-ETC-EUROASIC'94*, pages 25–40, March 1994.

[14] M. Sheliga, N.L. Passos, E.H. Sha. "Fully Parallel Hardware/Software Codesign for Multi-dimensional DSP Applications". In *Proceedings of 4th International Workshop on Hardware/Software Codesign*, pages 18–25, March 1996.

[15] N. Park, A.C. Parker. "Sehwa: A Software package for synthesis of pipelines from behavioral specifications ". In *IEEE Trans. on CAD*, volume 7, pages 356–370, March 1988.

[16] C-T. Hwang, Y-C. Hsu, Y-L. Lin. "Scheduling for functional pipelining and loop winding". In *Proc. Design Automation Conference*, pages 764–769, 1991.

[17] G. Goossens, J. Vandewalle, H.De Man. "Loop optimization in register-transfer scheduling for DSP systems". In *Proc. Design Automation Conference*, pages 826–831, 1989.

[18] T-F. Lee, A. C-H. Wu, Y-L. Lin, D.D. Gajski. "A transformation method for loop folding". In *IEEE Trans. on CAD*, pages 439–450, 1994.

[19] K. Parhi, D. Messerschmitt. "Static Rate Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding". In *IEEE Trans. on Computers*, volume 40 n2. pages 178–195, February 1991.

[20] C.E. Liserson, J.B. Saxe. "Optimizing Synchronous systems". In *Journal of VLSI an Computer Systems*, volume 1 n1. pages 41–67, Spring 1983.

[21] S.Y. Kung. *VLSI Array Processors.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[22] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* Mc Graw Hill, 1994.

Multicomponent Partitioning for VLSI System Synthesis

Nand Kumar and Ranga Vemuri

Address for Correspondence:

Dr. Ranga Vemuri, Director
Laboratory for Digital Design Environments
Department of Electrical and Computer Engineering
813 Rhodes Hall; Mail Location 30
University of Cincinnati
Cincinnati, Ohio 45221-0030

Phone: (513)-556-4784
Fax: (513)-556-7326
Email: ranga.vemuri@uc.edu

This work is done at the University of Cincinnati and is supported in part by the Solid State Electronics Directorate of the Wright Laboratory of the US Air Force under contract number F33615-91-C-1811 and by the Advanced Research Projects Agency (ARPAESTO RASSP Program) monitored by the US Air Force Wright Labs under contract no. F33615-93-C-1316

1 Introduction

Initial attempts at multicomponent synthesis involved carrying out high level synthesis and then partitioning the resultant design to realize a multichip design. High level synthesis converts a behavioral specification of a digital system into an equivalent RTL design (composed of a data path and a finite state controller; the data path is a composition of components selected from a register-level component library) that meets a set of stated performance constraints. This RTL design is then partitioned onto multiple chips to realize a multicomponent design. Recent efforts in system-level synthesis have led to the development of high level synthesis systems that can produce multichip digital systems [18, 46, 10]. These systems, however, do not consider the impact of packaging on high level synthesis and hence designs produced by these systems cannot efficiently use available high performance packaging technology.

Recent and ongoing revolution in electronics packaging has resulted in many high performance packaging technologies such as thin film multichip modules (MCMs). Packaging significantly impacts the performance and cost of systems. High level synthesis systems can no longer target just single chip designs or multichip designs without considering packaging technology. To make effective use of MCM technologies, high level synthesis systems must generate multichip structures taking into account the impact of packaging on system performance, heat, and cost.

Multicomponent Synthesis with Hierarchical Package Design is the process of high level synthesis targeting multichip and/or multicomponent implementations of the input behavioral specification to take advantage of available packaging technologies. Multicomponent synthesis and hierarchical package design is characterized by simultaneous synthesis of: (1) multiple register-level designs that interact with each other and together implement the function specified in the input behavioral specification; (2) a composition of these designs into a hierarchical structural design; and (3) a mapping of these register level designs and hierarchical structures onto efficient physical packages to realize a package hierarchy for the design.

Hierarchical RTL Partitioning and Package Design: Traditional partitioning and package design is restricted to a single level. A design is partitioned onto multiple packages at a particular level. However, digital designs occupy a hierarchy of packages from bare dies to boards (or backplanes and higher as needed). Also, packages come in various sizes with differing area and pin capacities and dollar costs. Cost effective packaging solutions for designs can be generated by carrying out hierarchical partitioning of the input RTL description onto a specified package library.

Payne and van Cleemput [38] developed an automatic partitioning technique for logic gates in order to meet gate and pin count constraints on chips. Beardslee et al [1] developed SLIP, an environment for system-level interactive partitioning. SLIP provides routines for maintaining and modifying a design hierarchy. These routines are used by partitioning algorithms to update and maintain design data. Saab and Rao [43] proposed an evolution based approach for partitioning logic circuits. Their approach takes constraints on the size of each part and number of pins. Also takes testable and critical nets into account during partitioning. Testable nets are cut to make them observable and critical nets are not cut.

Resnick designed SPARTA [39] to evaluate RTL designs with a spreadsheet-like approach. SPARTA checks for violation of area, power, and pin count constraints. Shih, Kuh, and Tsay [44] use a clustering step to satisfy timing constraints before using the Kernighan-Lin algorithm to partition functional blocks into a multicomponent design targeted to multichip modules (MCM). Vemuri applies genetic algorithms for partitioning register level designs for MCMs [47, 51]. A comparison with simulated annealing based partitioning is also presented.

Walker and Thomas [53] describe manual partitioning as part of design transformations in high level synthesis. McFarland [25] uses a hierarchical clustering technique, based on a measure of similarity, in partitioning behavioral hardware descriptions. These clustering algorithms are used in BUD [29] to perform a part of the allocation and module binding phase in data path synthesis in DAA [28]. Lagnese and Thomas

use a multistage clustering technique to partition a behavioral specifications into multiple processes to improve the quality of single chip designs [23, 24]. The approach shows significant area reductions in single chip designs, but does not consider design constraints or multichip implementations. Gupta and De Micheli [10] use the Kernighan-Lin and simulated annealing techniques for partitioning functional models while satisfying area and timing constraints. Pin-sharing or area/delay characteristic of registers, multiplexers, controllers, or wiring are not considered. Design constraints are not considered. Kucukcakar and Parker [17, 18] describe CHOP, a framework for interactive partitioning, in which the designer creates and modifies partitions and CHOP evaluates the validity of each partition by searching for possible implementations through predictions. Vahid and Gajski [46] describe partitioning at the algorithmic level. Clustering and Kernighan-Lin algorithms are used in partitioning. A preliminary bit-slice synthesis of behavioral objects in the design is performed prior to partitioning to generate performance characteristics of synthesized behavioral objects. Operator sharing across concurrent blocks is not considered — each concurrent block is synthesized separately and gets a set of dedicated hardware resources. During partitioning, as the composition of the design changes, new performance characteristics are not generated.

We develop a generic hierarchical graph partitioning and packaging model for (1) multicomponent synthesis with hierarchical package design and (2) hierarchical RTL partitioning and package design and propose a generic hierarchical partitioning and package design algorithm to accomplish the tasks. We present a generic input graph specification model for behavioral descriptions and RTL netlists (post high level synthesis) and a model for packaging options. We, then, formulate the hierarchical partitioning and package design problem and propose a solution. We, first, develop a mathematical model of the hierarchical partitioning and package design problem and, then, map our problem domains, (1) multicomponent synthesis with hierarchical package design and (2) partitioning register level designs onto a hierarchy of packages (from a package library), onto the mathematical model. We, then, propose a solution to the hierarchical partitioning and package design problem. We present experimental results for both approaches using our hierarchical partitioning and package design algorithm for some examples. And, finally, we present a comparison between multicomponent synthesis and hierarchical RTL partitioning and discuss the validity and applicability of our approach for modern designs and high performance packaging technologies.

2 Problem Formulation

An Example: Figure 1 shows an example graph. Consider the set of nodes of the graph, $N = \{n_1, n_2, n_3, n_4, n_5\}$. We shall use this example to illustrate some definitions in the problem formulation. Though we present the formulation for a generic graph, we discuss domain specific details for multicomponent synthesis and RTL partitioning as we present definitions.

The problem is introduced incrementally. Definitions 2.1 and 2.2 introduce the concept of a hierarchical k-level partition of a set. Definition 2.3 extends our notion of a k-level partition of a set to a k-level partition of a graph. Definition 2.4 defines a set of package levels. Definition 2.5 outlines a model for specifying package alternatives and their associated properties. Definition 2.6 shows the binding between a k-level partition of a graph and a set of package alternatives. The performance attribute computations are outlined in Definition 2.7.

Definition 2.1 A *1-level partition* of a set $\mathcal{N}$ is a system, $\mathcal{S}$, of nonempty sets (called segments) such that
(a) $\mathcal{S}$ is a system of mutually disjoint sets, i.e., if $C \in \mathcal{S}, D \in \mathcal{S}$, and $C \neq D$, then $C \cap D = \phi$,
(b) the union of $\mathcal{S}$ is the whole set $\mathcal{N}$, i.e., $\bigcup \mathcal{S} = \mathcal{N}$.

The set $S = \{s_1, s_2, s_3\}$, in Figure 1, defines a 1-level partition of N.
$s_1 = \{n_1, n_2\}$, $s_2 = \{n_3, n_4\}$, *and* $s_3 = \{n_5\}$.

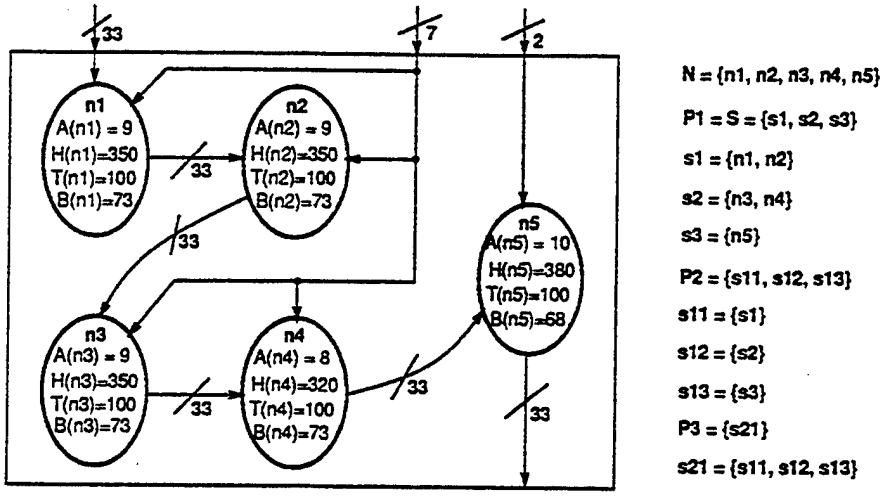


Figure 1: **An Example Graph and its k-level Partition**

Definition 2.2 A *k-level partition*, $\mathcal{P}$, of a set $\mathcal{N}$ is a set of 1-level partitions $P_1, P_2, \ldots, P_k$ such that
(a) for $1 \leq i \leq k-1$, P_{i+1} is a 1-level partition of P_i,
(b) P_1 is a 1-level partition of $\mathcal{N}$.

The 3-level partition of N (see Figure 1) is given by:
$P_1 = S = \{s_1, s_2, s_3\}$,
$P_2 = \{s_{11}, s_{12}, s_{13}\}$; $s_{11} = \{s_1\}$, $s_{12} = \{s_2\}$, *and* $s_{13} = \{s_3\}$, and
$P_3 = \{s_{21}\}$; $s_{21} = \{s_{11}, s_{12}, s_{13}\}$.

We extend the notion of a k-level partition of a set to define the k-level partition of a graph $G = (N, E)$, where N is the set of nodes and E is the set of edges. In the case of multicomponent synthesis, the input behavioral specification viewed as a process graph is the input graph, where N is the set of processes and E is the set of communication signals. In the case of RTL partitioning, the graph is the input RTL netlist, where N is the set of register level components and E is the set of interconnections between register level components.

Definition 2.3 A k-level partition of a graph $G = (N, E)$ is a k-level partition of N, where N is the set of nodes and E is the set of edges.
(a) area of a node $n \in N$ is given by $A(n)$,
(b) switching activity of a node $n \in N$ is given by $H(n)$,
(c) clock speed of execution of a node $n \in N$ is given by $T(n)$.

The performance attributes of nodes in the graph, $A(n)$, $H(n)$, and $T(n)$, are assumed to be primitive values supplied with the graph specification. In the case of multicomponent synthesis, performance attributes of nodes and level-1 partition segments in the graph, $A(n)$, $H(n)$, and $T(n)$, are determined through scheduling and performance estimation of individual nodes (level-1 partition segments) (see Section 3 and [19]). In the case of RTL partitioning, performance attributes of nodes are obtained from a register level component library. Only the area attribute of register level components is supported at the RTL level.

Definition 2.4 The *level set*, $\mathcal{L}$, is a set of k natural numbers $1, 2, \ldots, k$, i.e.,
$\mathcal{L} = \{1, 2, \ldots, k\}$.

Id	Area Capacity a(p) (sq mm)	Switch Capacity h(p)	Pin Capacity b(p)	Speed Capacity t(p) (ns)	Cost c(p) ($)	Level lmap(p)
p_1	5	400	40	50	400	1
p_2	10	400	80	50	600	1
p_3	18	1000	84	50	1500	1
p_4	6	600	40	50	250	2
p_5	12	600	80	50	300	2
p_6	20	1200	84	75	600	2
p_7	40	5000	64	100	200	3
p_8	60	5000	84	100	400	3

Table 1: **Example of Package Alternatives**

Definition 2.5
(1) P is a set of package alternatives, i.e., $P = \{p_1, p_2, \ldots, p_n\}$ with
area capacity $a(p)$, switching activity capacity $h(p)$, pin capacity $b(p)$, speed capacity $t(p)$, and cost $c(p)$ for $p \in P$
(2) *lmap* is a function that maps elements of P to the level set
$lmap : P \rightarrow \mathcal{L}$
(3) The minimal elements, P_{min}, of P are given by
$P_{min} = \{p \mid p \in P \text{ and } lmap(p) = 1\}$
(4) The maximal elements, P_{max}, of P are given by
$P_{max} = \{p \mid p \in P \text{ and } lmap(p) = k\}$
(5) A relation $\prec$ is defined in P such that
$p_1 \prec p_2$ iff package p_1 can be contained in package p_2, i.e.,
$lmap(p_2) = lmap(p_1) + 1$
(6) The *defining size* of a package set P is the package level of the maximal elements, i.e.,
$defining\ size = lmap(maximal\ element) = k$.

Table 1 shows an example set of package alternatives with area capacity, heat capacity, pin capacity, speed capacity, cost, and lmap defined for all its members. The defining size of this package set is three.

To realize a hierarchical package design, the k-level partition of a graph (Definition 2.3) needs to be bound to packages from the available set of package alternatives (Definition 2.5). Definition 2.6 describes this binding.

Definition 2.6 A *binding* of a k-level partition of a set $\mathcal{N}$ to a set of package alternatives P yields a set of map functions $\mathcal{M}$:
$\mathcal{M} = \{map_1, map_2, \ldots, map_k\}$
$map_i : P_i \rightarrow p_i$, $P_i \in \mathcal{P}$, $p_i \subset P$, p_i is a *bag*, i.e., duplicates are allowed in p_i and
$\forall p \in p_i\ lmap(p) = i$
such that
if S is a segment in P_i, then
$map_i(P_i) \succ map_{i-1}(S)$ i.e., $\forall p \in map_i(P_i)$ and $\forall q \in map_{i-1}(S)$, $p \succ q$.

Consider the 3-level partition of N from Figure 1. A *binding* of this 3-level partition of N to the set of package alternatives from Table 1 yields the following set of map functions:

$\mathcal{M} = \{map_1, map_2, map_3\}$,
$map_1 : P_1 \rightarrow \{p_3, p_3, p_2\}$,
$map_2 : P_2 \rightarrow \{p_6, p_6, p_5\}$, and
$map_3 : P_3 \rightarrow \{p_8\}$
$map_3(P_3) \succ map_2(P_2) \succ map_1(P_1)$, i.e.,
$lmap(p_6) = lmap(p_3) + 1$
$lmap(p_8) = lmap(p_6) + 1$.

To find a package design that satisfies constraints imposed by packages, rules of computation for performance attributes of partition segments need to be developed. Definition 2.7 outlines rules of computation to determine area, switching activity, pins, and speed of partition segments. Performance attributes of partition segments at higher levels of packaging are computed from performance attributes of constituent parts at lower packaging levels. Performance attributes of segments at level-1 are computed from primitive attributes of nodes in the input graph.

Definition 2.7 The computation rules for the physical attributes of area, heat, pins, and speed of a segment S in a 1-level partition P_i (part of a k-level partition $\mathcal{P}$) are defined below:

for $2 \leq i \leq k$:

(a) area of segment $A(S)$ is given by:

$$A(S) = \sum_{s \in S} a(map_{i-1}(s))$$

(b) heat of segment $H(S)$ is given by:

$$H(S) = \sum_{s \in S} H(s)$$

(c) pins of segment $B(S)$ are given by:

$$B(S) = \sum_{e_x \in E} e_x,\ e_x \text{ spans segments } s_a \text{ and } s_b;\ s_a \in S \text{ and } s_b \in S_y;\ S_y \in P_i,\ \text{and}\ S \neq S_y$$

(d) speed of segment $T(S)$ is given by:

$$T(S) = max(T(s)),\ s \in S;$$

for P_1:

(a) area of segment $A(S)$ is given by:

$$A(S) = \sum_{n \in S} A(n) \ \text{and}\ n \in N$$

(b) heat of segment $H(S)$ is given by:

$$H(S) = \sum_{n \in S} H(n) \ \text{and}\ n_j \in N$$

(c) pins of segment $B(S)$ are given by:

$$B(S) = \sum_{e_x \in E} e_x,\ e_x \text{ spans nodes } n_a \text{ and } n_b;\ n_a \in S \text{ and } n_b \in S_y;\ S_y \in P_1,\ \text{and}\ S \neq S_y$$

(d) speed of segment $T(S)$ is given by:

$$T(S) = max(T(n)),\ n \in S \ \text{and}\ n \in N.$$

In the case of multicomponent synthesis, performance attributes of level-1 partition segments are computed by carrying out a schedule and performance estimate step on each proposed segment. Physical attribute computation is shown below for the example in Figure 1.

for P_1:

$A(s_1) = A(n_1) + A(n_2) = 18$, $A(s_2) = A(n_3) + A(n_4) = 17$, and $A(s_3) = A(n_5) = 10$

$H(s_1) = H(n_1) + H(n_2) = 700$, $H(s_2) = H(n_3) + H(n_4) = 670$, and $H(s_3) = H(n_5) = 380$
$B(s_1) = 73$, $B(s_2) = 73$, and $B(s_3) = 68$
$T(s_1) = max(T(n_1), T(n_2)) = max(100, 100) = 100$,
$T(s_2) = max(T(n_3), T(n_4)) = max(100, 100) = 100$, and
$T(s_3) = max(T(n_5)) = 100$

for P_2:
$A(s_{11}) = a(map_1(s_1)) = a(p_3) = 18$, $A(s_{12}) = a(map_1(s_2)) = a(p_3) = 18$, and
$A(s_{13}) = a(map_1(s_3)) = a(p_2) = 10$
$H(s_{11}) = H(s_1) = 700$, $H(s_{12}) = H(s_2) = 670$, and $H(s_{13}) = H(s_3) = 380$
$B(s_{11}) = 73$, $B(s_{12}) = 73$, and $B(s_{13}) = 68$
$T(s_{11}) = max(T(s_1)) = 100$, $T(s_{12}) = max(T(s_2)) = 100$, and $T(s_{13}) = max(T(s_3)) = 100$

for P_3:
$A(s_{21}) = a(map_2(s_{11})) + a(map_2(s_{12})) + a(map_2(s_{13})) = 20 + 20 + 12 = 52$
$H(s_{21}) = H(s_{11}) + H(s_{12}) + H(s_{13}) = 1750$
$B(s_{21}) = 75$
$T(s_{21}) = max(T(s_{11}), T(s_{12}), T(s_{13})) = 100$.

Definition 2.8 formulates the hierarchical package design problem for an input graph G and a package set P. The hierarchical k-level package design problem is presented below as a constraint satisfying k-level partitioning problem (Definition 2.3) that is bound to packages from the package library. At each level, i in the package hierarchy, the binding generated by map_i has to be a package from the set of packages such that performance constraints are satisfied. Also, cost constraint on the entire design has to be satisfied.

Definition 2.8 Given $G = (N, E)$, a package set P with defining size k, and a cost constraint C, find a k-level partition $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ of G and a binding of $\mathcal{P}$ to P such that
for $1 \leq i \leq k$, if $S \in P_i$

$A(S) \leq a(map_i(S))$,
$H(S) \leq h(map_i(S))$,
$B(S) \leq b(map_i(S))$,
$T(S) \geq t(map_i(S))$.

subject to

$$Cost(\mathcal{P}) = \sum_{i=1}^{k} c(map_i(P_i)); \quad Cost(\mathcal{P}) \leq C.$$

A cost constraint of \$ 5500.00 yields a solution to the k-level partitioning problem, for our running example (Figure 1), with cost \$ 5500.00 and the following characteristics of the binding (see Figure 2).

for P_1:
$(A(s_1) = 18) \leq (a(map_1(s_1)) = a(p_3) = 18)$, $(A(s_2) = 17) \leq (a(map_1(s_2)) = a(p_3) = 18)$, and
$(A(s_3) = 10) \leq (a(map_1(s_3)) = a(p_2) = 10)$
$(H(s_1) = 700) \leq (h(map_1(s_1)) = h(p_3) = 1000)$, $(H(s_2) = 670) \leq (h(map_1(s_2)) = h(p_3) = 1000)$, and
$(H(s_3) = 380) \leq (h(map_1(s_3)) = h(p_2) = 400)$
$(B(s_1) = 73) \leq (b(map_1(s_1)) = b(p_3) = 84)$, $(B(s_2) = 73) \leq (b(map_1(s_2)) = b(p_3) = 84)$, and
$(B(s_3) = 68) \leq (b(map_1(s_3)) = b(p_2) = 80)$
$(T(s_1) = 100) \geq (t(map_1(s_1)) = t(p_3) = 50)$, $(T(s_2) = 100) \geq (t(map_1(s_2)) = t(p_3) = 50)$, and
$(T(s_3) = 100) \geq (t(map_1(s_3)) = t(p_2) = 50)$

for P_2:
$(A(s_{11}) = 18) \leq (a(map_2(s_{11})) = a(p_6) = 20)$, $(A(s_{12}) = 18) \leq (a(map_2(s_{12})) = a(p_6) = 20)$, and

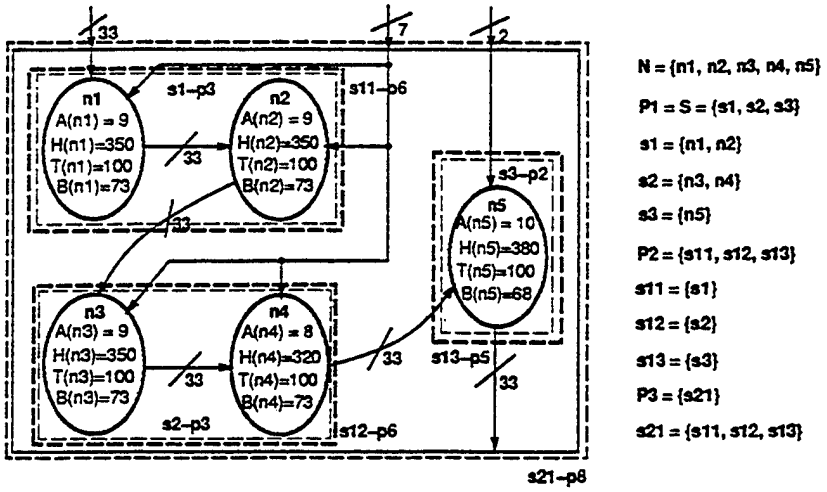


Figure 2: Example Solution

$(A(s_{13}) = 10) \leq (a(map_2(s_{13})) = a(p_5) = 12)$
$(H(s_{11}) = 700) \leq (h(map_2(s_{11})) = h(p_6) = 1200)$, $(H(s_{12}) = 670) \leq (h(map_2(s_{12})) = h(p_6) = 1200)$, and
$(H(s_{13}) = 380) \leq (h(map_2(s_{13})) = h(p_5) = 600)$
$(B(s_{11}) = 73) \leq (b(map_2(s_{11})) = b(p_6) = 84)$, $(B(s_{12}) = 73) \leq (b(map_2(s_{12})) = b(p_6) = 84)$, and
$(B(s_{13}) = 68) \leq (b(map_2(s_{13})) = b(p_5) = 80)$
$(T(s_{11}) = 100) \geq (t(map_2(s_{11})) = t(p_6) = 75)$, $(T(s_{12}) = 100) \geq (t(map_2(s_{12})) = t(p_6) = 75)$, and
$(T(s_{13}) = 100) \geq (t(map_2(s_{13})) = t(p_5) = 50)$

for P_3:
$(A(s_{21}) = 52) \leq (a(map_3(s_{21})) = a(p_8) = 60)$
$(H(s_{21}) = 1750) \leq (h(map_3(s_{21})) = h(p_8) = 5000)$
$(B(s_{21}) = 75) \leq (b(map_3(s_{21})) = b(p_8) = 84)$
$(T(s_{21}) = 100) \geq (t(map_3(s_{21})) = t(p_8) = 100)$

$$Cost(\mathcal{P}) = \sum_{i=1}^{k} c(map_i(P_i)) = c(map_1(P_1)) + c(map_2(P_2)) + c(map_3(P_3)) = \$\,5500$$

$c(map_1(P_1)) = c(p_3) + c(p_3) + c(p_2) = \$\,3600$
$c(map_2(P_2)) = c(p_6) + c(p_6) + c(p_5) = \$\,1500$
$c(map_3(P_3)) = c(p_8) = \$\,400$

Cost of packaging $Cost(\mathcal{P})$ is \$ 5500 and cost constraint C is \$ 5500.
Thus, $Cost(\mathcal{P}) \leq C$.

3 Scheduling and Performance Estimation

Scheduling and *Performance Estimation* are important steps in high level synthesis and are used to explore the design space [3, 27, 17, 18]. We briefly describe scheduling (see [35, 36, 37, 6] for more details) and performance estimation (see [11, 12, 13, 30, 21, 20, 19] for more details).

Scheduling: Scheduling is the first important step in the synthesis process. The input behavioral specification is converted into an equivalent data flow graph (DFG) representation. Scheduling operates on the DFG. DFG operations are assigned to specific control steps and are bound to physical ALUs available in

the component library. The output of scheduling is a time-stamped and partially bound data flow graph, that satisfies user specified constraints. Scheduling determines execution speed of the synthesized design in terms of clock speed and number of clock cycles required to execute all operations. In addition, it fixes control and data path (ALU) architectures — the architecture impacts on performance of the design. An implementation of Paulin's *force-directed list scheduling* [35, 36], extended for communicating and concurrently executing processes [6], is used. Force-directed scheduling produces maximally fast (minimum number of control steps) schedules under resource constraints. Force-directed scheduling tries to maximize operation concurrency, ensuring high resource utilization. Hardware resources are shared across concurrent blocks. As a result, operations in concurrent blocks are scheduled under global resource constraints. All operations are treated as macro operations that execute in one logical control step. Operations such as '+', '-', and *call* etc. are treated alike. Logical control steps are expanded into equivalent physical clock steps during control generation [41]. All arithmetic, logical, and relational operations engage a single hardware resource. Subprograms, loop, and wait modules are assumed to engage all available resources. Hence, call operations do not share control steps with any other operation, i.e., no other operation is scheduled in the same control step as a call operation.

Performance Estimation: For high performance packaging technology such as MCMs, power/heat dissipation in the design is very important. An accurate performance estimator for power/heat dissipation is needed to generate good designs. Many studies in power estimation for switch level and gate level circuits have assumed that average power dissipation is directly proportional to the average switching activity [32, 31, 15, 4, 45, 2, 42]. In CMOS designs, dynamic power consumption is predominant and is directly proportional to the aggregate (total) switching activity (ASA) in the circuit. ASA in the design is defined as the total number of circuit node switchings and is dependent upon the input patterns stimulating the circuit. The design is composed of components from a cell library and a finite state controller implemented as a collection of PLAs.

We use a profile-driven approach to switching activity estimation. In this approach, event activities related to various operations and carriers in the behavioral specification are measured by simulating the description using user-supplied inputs. A *profiler* is a tool that simulates the behavioral specification with user-supplied input patterns, called *profiling stimuli.* Before simulation begins, the profiler alters the behavioral specification by inserting probes (counters) to monitor *event activity* in various regions of the specification. At the end of simulation, the profiler prints the number of times each statement is executed, number of invocations of each function and similar data pertaining to the event activity that occurred in the behavioral specification during the simulation run. These event activities are then used during the synthesis process (during *performance estimation*) to estimate the switching activity in the design being synthesized.

High level synthesis uses a library of parameterized register level module generators. Modules are parameterized with respect to number of inputs where applicable and bit-width of each input. The library contains interface descriptions of each module, description of its parameters, and its area, delay and average intrinsic switching activity (ISA) characteristics. Area, delay and ISA values of each library module are determined by actually generating layouts for several instances of the module with different parameter values. Determination of area and delay parameters for layout instances is straightforward. Area can be directly measured from the layout and delay can be determined through simulation or a timing analysis program such as Crystal [34]. We define the *average intrinsic switching activity* (ISA) of a module instance as the average number of circuit nodes that are expected to switch when an input event (change of logic values on the input lines) takes place. ISA of a module instance is determined by extracting a switch level model from its layout, simulating the switching level module using a very long stream of randomly generated input patterns and counting the average number of circuit nodes switched per pattern. Simulation and counting continues until convergence occurs.

Overall switching activity estimation is based on using event activities to modulate the average intrinsic switching activities of library modules used in the synthesis process. This estimate is used to, in addition

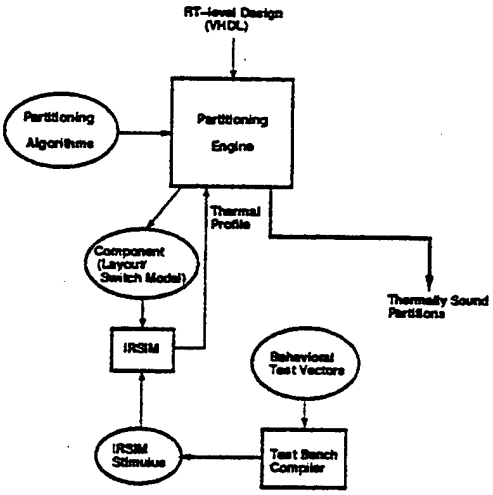


Figure 3: **Thermal Profiling of RT level Components**

to area and clock-speed estimates, to guide the synthesis process. Experimental results for a number of examples show that switching activity estimated during synthesis deviates by less than 10% on the average from the actual switching activity measured after completing synthesis [20]. Area and delay estimation are based on the work of Jain [11], Kurdahi [21], Mlinar [30], and Dutta [6].

Thermal Profile of RT level components: For modern high performance packaging technology such as multichip modules (MCMs) heat dissipation in the design is a critical performance measure. For efficient utilization of such high performance packaging technologies, thermal constraints of packages need to be satisfied. To evaluate the feasibility of partitions, accurate power/heat dissipation figures of the register level components is required by the partitioning algorithms. Power/heat dissipation can be approximated by an estimation of switching activity in a design as average power dissipation in a circuit is directly proportional to the average switching activity. The switching activity estimation procedure consists of counting the switching activity of nodes in a circuit during a switch level simulation of layout/switch level models of the register transfer level components with a characteristic set of test vectors. A characteristic set of test vectors for each component is derived from the set of behavioral test vectors used by the designer to validate the behavioral specification prior to synthesis.

Figure 3 demonstrates the technique of switching activity estimation. Partitions with single register level components are generated. Each register level component in the synthesized design is placed on a separate partition. Layouts and switch level models of these single component partitions are generated. The switch level models are simulated with switch level test benches (generated using a *test bench compiler* – TBC [49, 52]) and the number of nodes switching in the switch level model are counted. This count of node switches gives a very accurate measure of the power/heat dissipation in the register level component. This switching activity data is used by the partitioning algorithms to generate thermally sound partitions.

This process of generating switching activity measures for all register level components in the RTL design is too time consuming. For example, a small traffic light controller example (TLC, see [19, 48] has 49 RTL components and gets synthesized to a 4769 transistor design. Five behavioral test vectors get translated into 1320 switch level test vectors (for each component). Complete layout generation, extraction of switch level models, conversion of behavioral test vectors into switch level test vectors, and switch level simulation together took about 48 hours. The layouts and switch level models of every RTL component needs to be generated individually and each of them has to be simulated at the switch level with a characteristic set of input vectors. A handful of test vectors at the behavioral level explode into thousands of switch level

vectors. Layout generation and switch level simulation (for all components) are too time consuming for this technique of switching activity estimation to be viable for large RTL designs.

4 Hierarchical Partitioning and Package Design Algorithm

The solution to the above problem takes the form of a *hierarchical partitioning and package design algorithm* that incorporates back-tracking while tightening cost constraints on the design with each succeeding refinement in the design. The algorithm takes the following inputs: (1) behavioral VHDL specification of a digital system viewed as a process graph composed of communicating and concurrently executing processes; alternately, an RTL netlist composed of register level components; (2) parameterized register level component library characterized for area, delay, and switching activity; (3) package library with area, pins, switching activity, clock speed, and cost information for all packages; (4) cost constraint C, in dollars on the entire design. The algorithm begins by partitioning the process graph and mapping partition segments (after scheduling/performance estimation to obtain accurate performance attributes of the design) onto available bare-die packages; alternately, by partitioning the RTL netlist and mapping segments in the partition onto available bare-die packages. A graph is constructed from the generated partition at this level for further partitioning at the next higher level of packaging. The packaged partition segments form nodes in the new graph; edges of the graph are obtained from the interconnection of register level designs in the multicomponent design. At the next higher level of packaging, this new graph is partitioned and mapped onto packages. This process continues until the packaging hierarchy is exhausted and at each level, partition segments are mapped onto cost effective packages. If, at a particular level, no solution is found, we back-track to the previous level, tighten cost constraints, and construct a new partition and continue.

The output of hierarchical partitioning and package design is: (1) a set of RTL designs (individual RTL designs that together form the multicomponent design); (2) a set of structures that realizes the hierarchical design; and (3) a binding of the RTL designs and structures to appropriate cost effective packages from the package library at each level of packaging. The design satisfies capacity constraints imposed by packages and the algorithm composes designs and picks packages such that overall cost constraint on the design is satisfied.

Partitioning and package design at each level involves: (1) determining cost constraint and physical constraints on the design — overall area and switching activity constraints on the design are derived from the minimum capacity package at the highest level in the package hierarchy (say, the minimum area and switching activity capacities of all available boards if the package hierarchy ended at board level); individual segment area, switching activity, clock speed, and pin constraints are derived from the capacity of available packages at a particular level of packaging; (2) constructing the partition subject to constraints and mapping onto a set of cost effective packages. At level-1 in the packaging hierarchy, in the case of multicomponent synthesis, scheduling and performance estimation is carried out on each proposed partition segment and performance attributes of the segment are determined and feasibility of the multicomponent design and partition checked. At other levels in the packaging hierarchy, performance attributes of proposed partition segments are composed of its constituent parts and their packaging; (3) checking to see if constraints are satisfied and if we need to back-track or proceed to next higher level of packaging; and (4) construct netlist for next level and go to (1). At any level in the package hierarchy, the *cost constraint* is determined by deducting the cost of packaging partitions at lower packaging levels and the projected cost at higher packaging levels from the total cost constraint, C.

Setting Constraints: Initially, on the first pass, overall constraints on area and switching activity constraints on the entire design are derived from the minimum area and switching activity capacity of packages at the highest level in the package hierarchy (since, eventually, the design hierarchy needs to be mapped onto a package at the top level in the package hierarchy); the cost constraint is set by subtracting the

Algorithm 4.1 (Set_Constraint)

P: package set, C: overall cost constraint on design
k: levels in package hierarchy, level: current level
area: overall area constraint, cost: cost constraint at current package level
CTF: constraint tighten factor (< 1), COF: cost overrun factor (< 1)
pass: flag to generate physical constraints on design (initially 1)

Set_Constraint()
begin
  if *pass = 1* **then** /* *set physical constraints from package at level k in package hierarchy* */
    $cost \leftarrow C - \sum_{l=2}^{k}$ *smallest package cost*
    area $\leftarrow$ *min(area capacity of package at level k)*
    switch $\leftarrow$ *min(switching activity capacity of package at level k)*
    pass $\leftarrow$ *pass* + *1* /* *set flag to indicate physical constraints set* */
  elsif *(status = SUCC)* $\vee$ *(b_track = FALSE)* **then**
    $cost \leftarrow C - \sum_{l=level+1}^{k}$ *smallest package cost* $- \sum_{l=1}^{level-1}$ *package cost*
  elsif *b_track = TRUE* **then**
    $cost_over_run \leftarrow cost_{level} - cost$
    if $cost_over_run < cost_{level-1}$ **then**
      $cost \leftarrow cost_{level-1} - cost_over_run \times COF$
    else
      $cost \leftarrow cost_{level-1} \times CTF$
    end if
  end if
end

cost of the smallest packages at all levels of packaging above level-1. On subsequent invocations, if the algorithm is back-tracking, a cost overrun is computed; if the cost overrun is less than the cost of the previous level's packaging, cost constraint for the previous level (on a back-track) is set by subtracting a product of the cost overrun and a *cost overrun factor* (COF < 1) from the cost of the previous level's packaging; if the cost overrun is greater than the cost of the previous level's packaging, cost constraint for the previous level (on a back-track) is set by multiplying the cost of the previous level's packaging by a *constraint tighten factor* (CTF < 1). COF and CTF dictate the rate at which the cost constraint is tightened on a back-track. Typical values of COF are between 0.2–0.3 and CTF between 0.9–0.95 to enable effective search of the design space. If the algorithm is not back-tracking, cost constraint is generated by subtracting the actual cost of packaging at lower levels of packaging and the projected packaging cost at higher levels (cost of smallest packages) from the total cost constraint, C.

Hierarchical Partitioning and Package Design (HPP): Algorithm 4.2 presents the hierarchical partitioning and package design algorithm (HPP). HPP has access to a hierarchical clustering based partitioning algorithm (HCP – Algorithm 4.3) and a multiway partitioning algorithm (MP – Algorithm 4.4). When partitioning at any level, HPP first determines cost, area, and switching activity constraints using Set_Constraint (Algorithm 4.1). HPP then invokes HCP to generate a partition and a binding of its partition segments to packages from the package library. HCP utilizes the underlying clustering in the design to quickly generate a partition. If HCP does not find a constraint satisfying solution, MP is invoked. MP explores a larger design space by constructing a class of partitions; MP returns the first partition that satisfies constraints, or, in the absence of a constraint satisfying solution, returns the best cost solution from the class of partitions.

Algorithm 4.2 (HPP Algorithm: HierPartPack)

G: input graph (Behavioral specification/RTL netlist), P: package set
C: overall cost constraint on design, HN: hierarchical netlist manager
StatArr[k], BtkArr[k]: status of partitioning and number of back-tracks at each level
MaxBtk: User specified limit on number of back-tracks at any level
k: levels in package hierarchy, level: current level, area: overall area constraint
switch: overall switch activity constraint, cost: cost constraint at current package level

```
HierPartPack(G, P, C)
begin
  level ← 1      G_level ← G      Solution ← null
  while level < k do
     Set_Constraint()
     (HcpStatus, HcpSolution) ← HCP(G_level, P(level), cost, area, switch, level)
     if HcpStatus ≠ SUCC then
        (status, Solution) ← MP(G_level, P(level), cost, area, switch, level)
     end if
     if (status ≠ SUCC) ∧ (cost(HcpSolution) < cost(MpSolution)) then
        (status, Solution) ← (HcpStatus, HcpSolution)
     end if
     StatArr[k] ← status
     case status is
        SUCC:
           level ← level + 1          HN :: read_partition(Solution)
           HN :: construct_netlist(level)      /* construct netlist at next level */
        BEST:
           if (StatArr[level − 1] = SUCC) ∧ (BtkArr[k] < MaxBtk) then
              BtkArr[k] ← BtkArr[k] + 1          level ← level − 1     /* back-track */
           else
              level ← level + 1          HN :: read_partition(Solution)
              HN :: construct_netlist(level)
           end if
        FAIL:
           if (StatArr[level − 1] = SUCC) ∧ (BtkArr[k] < MaxBtk) then
              BtkArr[k] ← BtkArr[k] + 1          level ← level − 1     /* back-track */
           else      return(null)      end if
     end case
     G_level ← HN :: read_netlist(level)          /* retrieve next level netlist */
  end while
  return(Solution)
end
```

Both partitioning algorithms, HCP and MP, return a *status* along with a solution (partition with segments bound to packages). Status takes three values of *SUCC, BEST,* or *FAIL* to describe the cases where a constraint satisfying solution is found (a constraint satisfying partition with partition segments mapped onto packages from the package library), a solution is found (valid partition – a partition with segments mapped onto packages), or no solution is found (no valid partition – one or more partition segments cannot be mapped onto packages).

Status is used to decide the execution flow of HPP. If the status of partitioning is SUCC, then HPP proceeds to the next higher level of packaging. A *hierarchical netlist manager* (HN) is used to generate a netlist, of the newly generated partition, for use at the next higher level. If, at a particular level, the status is BEST or FAIL, and: if the previous level partition's status is SUCC, HPP back-tracks to the previous level and generates a new partition with tighter cost constraints; if the previous level partition's status is BEST and the current level partition's status is BEST, HPP proceeds to the next higher level of packaging; if the previous level partition's status is BEST and the current level partition's status is FAIL, HPP terminates reporting failure to find a solution. HN is used to generate the netlist for partitioning. On a recursive back-track, back-tracking continues until we reach a level where the status of partitioning is BEST. When we encounter a status of BEST, we cannot do any better and the back-track stops, and the algorithm proceeds to the next higher level of packaging.

Hierarchical Cluster-based partitioning (HCP): Hierarchical clustering is the partitioning technique [14]. Algorithm 4.3 describes HCP. A cluster tree for the input graph is constructed using the hierarchical clustering approach. The hierarchical clustering algorithm groups a set of objects according to some measure of closeness [14]. Two closest objects are clustered first and considered to be a single object for future clustering. Clustering continues by grouping two individual objects, or an object or cluster with another cluster on each iteration. The process stops when a single cluster is generated and a hierarchical cluster tree is formed. Alternate partitions are constructed by traversing this cluster tree and moving the cut-line [14, 25, 23, 24]. Figure 4 shows an example cluster tree and the different cut-lines and associated partitions. A *map* function maps partition segments to available packages in the package library. Partition segments and the entire partition are then checked for constraint satisfaction. A sum of package costs (for all partition segments) gives the cost of the partition. In the case of a constraint satisfying solution (performance and cost), the solution (partition) is returned to the hierarchical partitioning algorithm with status SUCC. In the case of a solution (valid partition with partition segments mapped onto packages) that does not satisfy constraints, a status BEST is returned. When no solution (no valid partition – one or more partition segments cannot be packaged) is found, a FAIL is returned.

Multiway Partitioning Algorithm (MP): MP (Algorithm 4.4) is built on top of the Multiway Fiduccia-Mattheyses algorithm (MFM — Algorithm 4.5). MP first determines the minimum and maximum number of segments that *feasible* partitions can have (the partition is feasible, i.e., there may exist a partition such that partition segments can be effectively bound to packages from the package library). The minimum number of segments (*min_seg*) is determined by; dividing the area constraint on the design by the area capacity of the largest package; dividing the switching activity constraint on the design by the switching activity capacity of the largest package; and picking the larger of the two. The maximum number of segments (*max_seg*) is determined as the number of nodes in the input graph (in the case of multicomponent synthesis, the number of processes in the input process graph; alternately, a user specified limit on the number of RTL components or the number of RTL components in the case of an RTL netlist). MP invokes MFM to generate partitions with number of segments varying from min_seg to max_seg. MP returns with status SUCC if a constraint satisfying partition is found. When a constraint satisfying solution is not found, MP returns the best solution found with status BEST. In the case of no valid partitions (one or more partition segments cannot be packaged), MP returns FAIL. Algorithm 4.5 presents the modified MFM algorithm. MFM repeatedly calls a K-way Ficuccia-Mattheyses based partitioning algorithm (KWAY – Algorithm 4.6) to generate partitions. MFM keeps track of the best cost partition. MFM returns a constraint satisfying partition, if found, or the best cost partition. KWAY determines area, switching activity, clock speed, and

Algorithm 4.3 (HCP Algorithm)

CD : depth of cluster tree, P : partition at current level in cluster tree
S : segment in partition, p : package segment is mapped to
level : level in the package hierarchy, cost : cost constraint on current package level
area : overall area constraint, switch : overall switching constraint

```
HCP(level, G, PackageLib, cost, area, switch)
begin
  construct cluster tree (T)
  best_cost ← ∞      Solution ← null      status ← FAIL      CD ← depth(T)
  for tree_level = 1 to CD do
     constraint_satisfied ← TRUE
     for each S ∈ P do      /* individual partition segment constraints */
        if level = 1 then      /* pure behavior specification  - -  estimate attributes */
           Schedule/Performance Estimate S and generate A(S), H(S), B(S), and T(S)
        end if
        p ← PackageLib :: map(S)      /* get package segment S fits on */
        if ((p ≠ null) ∧ (A(S) < A(p)) ∧ (H(S) < H(p)) ∧ (B(S) < B(p)) ∧
            (T(S) ≥ T(p)))  then      constraint_satisfied ← constraint_satisfied ∧ TRUE
        else      constraint_satisfied ← FALSE      end if
     end for
     if constraint_satisfied = TRUE then      /* overall design constraints */
        if ((cost(P) < cost) ∧ (Area(P) ≤ area) ∧ (Switch(P) ≤ switch)) then
           return (SUCC, P)
        elsif cost(P) < cost then
           Solution ← P      cost ← cost(P)      status ← BEST
        end if
     end if
  end for
  return (status, Solution)
end
```

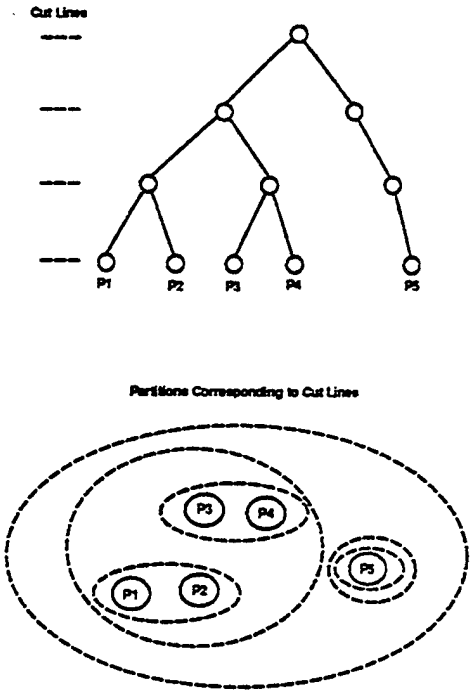


Figure 4: **Example Cluster Tree and Partitions**

pin constraints from packages in the package library and using these constraints generates a partition and maps partition segments to packages. At the completion of KWAY, the algorithm returns a partition with segments bound to packages in the package library.

k-way Fiduccia-Mattheyses Algorithm (KWAY): The *k-way* FM algorithm (KWAY — Algorithm 4.6) starts by creating a random initial partition of n partition segments. Nodes in the graph are randomly assigned to the n segments. Each segment gets some nodes from the set of vertices V of the graph G. The initial partition is saved in *Best*. Cost of this partition is saved as *best_cost*. k-way partitioning is carried out by repeatedly invoking *two-way* FM (*two_way_fm*) on pairs of partition segments. *two_way_fm* tries to improve bi-partitions by moving one node at a time from one partition segment to the other, taking care not to violate area and switching activity constraints. The *two_way_fm* algorithm is based on Fiduccia and Mattheyses's bi-partitioning algorithm [8]. *two_way_fm* is invoked until, either a user specified limit on number of total iterations is exceeded, or a user specified limit on number of iterations over which partition cost does not improve is exceeded. The best cost solution found during the iterations is returned as the k-way partition.

Multicomponent Synthesis: Multicomponent synthesis is carried out when the input is a behavioral specification. HCP and MP algorithms carry out multicomponent synthesis at level-1 in the package hierarchy. Multicomponent synthesis is carried out by synthesizing individual partition segments at level-1. Design tradeoffs are performed by considering various partitions and carrying out scheduling and performance estimation on proposed partition segments and determining performance attributes of the synthesized RTL designs and determining if they satisfy capacity and cost constraints imposed by available packages. Also, a *global controller* is automatically placed on a partition segment and interconnected with the RTL design segments. The global controller is placed on a partition segment whose package has the most space to fit the controller. HCP (Algorithm 4.3) considers different partitions by traversing the cluster tree — each level in the cluster tree represents a different partition (see Figure 4). At level-1, every time a new partition is considered — HCP carries out scheduling and performance estimation on each of the proposed partition

Algorithm 4.4 (Multiway Partitioning Algorithm: MP)

G: input graph, P: package set, p: individual package from P
area: overall area constraint, switch: overall switch activity constraint
C: cost constraint on design, level: level in package hierarchy

```
MP(G, P, C, area, switch, level)
begin
  min_seg ← max(area ÷ max_area(p), switch ÷ max_switch(p))
  max_seg ← num_cell(G)      /* number of nodes in graph */
  best_cost ← ∞      status ← FAIL      Solution ← null
  for num_seg = min_seg to max_seg do
     (status, TempSolution) ← MFM(G, P, num_seg, C, area, switch, level)
     if status = SUCC then
        return (status, TempSolution)
     elsif (status = BEST) ∧ (cost(TempSolution) < best_cost) then
        Solution ← TempSolution          best_cost ← cost(TempSolution)
     end if
  end for
  return(status, Solution)
end
```

segments (to compute performance attributes of the RTL design) and then tries to map these segments onto packages from the package library. Multicomponent synthesis in MP occurs in KWAY (Algorithm 4.6). At level-1 whenever a new partition is constructed, scheduling and performance estimation are carried out on individual partition segments. In Algorithm 4.6 a schedule/performance estimate step is carried out when the initial partition is generated and also every time a new partition is generated. By scheduling and performance estimation, we predict the performance characteristics of the individual synthesized RTL designs and also of the entire multicomponent design.

At the end of multicomponent synthesis and hierarchical package design we have a multicomponent design composed of interacting RTL design segments — the multicomponent synthesis phase produces multiple behavior segments that are completely synthesized to RTL designs using a high level synthesis system such as DSS [40, 41]. Also produced is a hierarchical structural design (the leaf nodes in this design are the individual RTL designs) that is mapped onto efficient cost-effective packages from a package library.

An Example: We illustrate the HPP algorithm (Algorithm 4.2) through an example. The graph in Figure 1 is partitioned onto the package set specified in Table 1 (to generate a hierarchical design that is mapped onto cost effective packages). Hierarchical partitioning and package design generates a package hierarchy in addition to a multichip design for the input specification.

Let the user specified cost constraint, C, be \$ 5000. First the overall area and overall switching activity constraints are determined from the capacity of the smallest package at the highest package level (since, eventually, the design hierarchy will be mapped onto a package at the highest level in the package hierarchy) — the overall area and switching activity constraints are set from the area and switching activity capacities of p_7 at level-3 which has an area capacity of 60 sq mm and switching activity capacity of 5000. The cost constraint on level-1 packaging is given by subtracting the projected packaging costs at levels 2 and 3 from C, i.e., by subtracting the cost of the smallest packages at each of these levels from C. The cost constraint on level-1 packaging is \$ 4550 (5000 – 200 – 250). *Set_Constraint* (Algorithm 4.1) is used to set the area

Algorithm 4.5 (Multiway Fiduccia-Mattheyses Algorithm: MFM)

G: input graph, P: package set, C: cost constraint on design
A: overall area constraint, S: overall switching activity constraint
n: number of segments, level: level in package hierarchy

```
check_constraint(S)
begin
  status ← BEST    tot_area ← 0    tot_switch ← 0    tot_cost ← 0
  for all s_i ∈ S do        /* segments in partition */
     if map(s_i) = null then        /* partition segment not mapped to package */
        return(FAIL)
     end if
     tot_area ← tot_area + area(map(s_i))
     tot_switch ← tot_switch + switch(s_i)
     tot_cost ← tot_cost + cost(map(s_i))
  end for
  if (tot_area ≤ A) ∧ (tot_cost ≤ C) then
     status ← SUCC
  end if
  return(status)
end

MFM(G, P, n, C, A, S, level)
begin
  Best ← KWAY(G, P, n, level)    /* generate first partition */
  num_fm_ite ← 1    num_fm_imp ← 1    status ← check_constraint(Best)
  while status ≠ SUCC ∧ num_fm_ite < MAX_FM_ITE ∧ num_fm_imp < MAX_FM_IMP do
     S ← KWAY(G, P, n, level)  status ← check_constraint(S)  num_fm_ite ← num_fm_ite + 1
     best_cost ← cost(Best)
     if (status = SUCC) ∨ ((status = BEST) ∧ (cost(S) < best_cost)) then
        Best ← S
     end if
     if (cost(S) < best_cost) then  num_fm_imp ← 1
     else    num_fm_imp ← num_fm_imp + 1    end if
  end while
  return(status, Best)
end
```

Algorithm 4.6 (k-way FM Algorithm: KWAY)

G: graph $G = (V, E)$, V is a set of vertices and E is a set of edges
P: set of packages, S: $\{s_1, s_2, \cdots, s_n\}$ a partition of G with n segments

```
KWAY(G, n, level)
begin
  Best ← initialize()      /* create initial partitions */
  if level = 1 then    /* pure behavior specification -- estimate attributes */
     for all s ∈ Best do
        Schedule/Performance Estimate s and generate A(s), H(s), B(s), and T(s)
     end for
  end if
  best_cost ← 0     S ← null     cont_part ← TRUE     ite_cnt ← 1     imp_cnt ← 1
  for all s ∈ Best do     /* map partition segment to package and find cost */
     best_cost ← best_cost + cost(map(s))
  end for
  while cont_part = TRUE do
     for i = 1 to n−1 do
        for j = i+1 to n do
           two_way_fm(s_i, s_j)
        end for
     end for
     if level = 1 then    /* pure behavior specification -- estimate attributes */
        for all s ∈ S do
           Schedule/Performance Estimate s and generate A(s), H(s), B(s), and T(s)
        end for
     end if
     curr_cost ← 0
     for all s ∈ S do     /* map partition segment to package and find cost */
        curr_cost ← curr_cost + cost(map(s))
     end for
     ite_cnt ← ite_cnt + 1
     if curr_cost < best_cost then
        imp_cnt ← 1     Best ← S          /* save best partition seen so far */
     else   imp_cnt ← imp_cnt + 1   end if
     if ite_cnt = MAX_ITE ∨ imp_cnt = IMP_CNT then   cont_part ← FALSE   end if
  end while
  return(Best)          /* retrieve best partition */
end
```

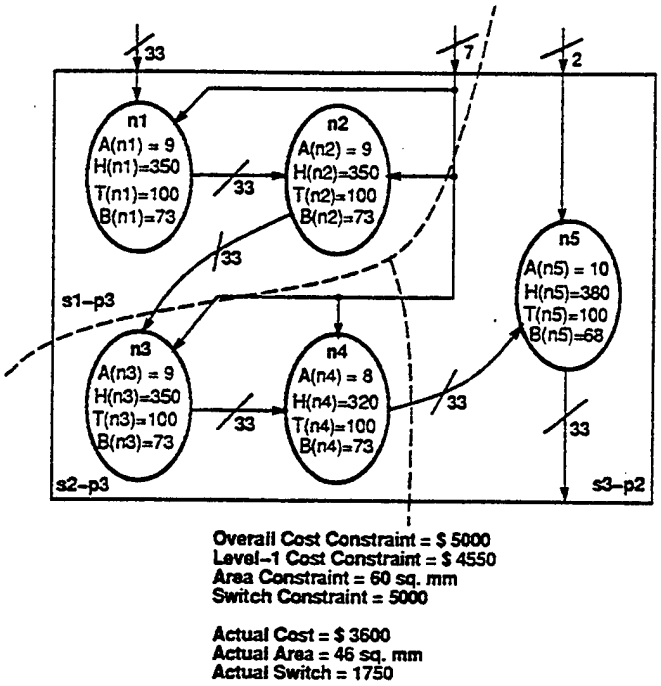


Figure 5: **Level-1 Partition — First Pass**

and switching activity constraints on the entire design and the cost constraint for level-1.

Having determined the cost, area, and switching activity constraints on the partition at level-1, the next step is to construct a partition. HPP invokes HCP to generate a partition and a binding of its partition segments to packages from the package library. If HCP does not find a constraint satisfying solution, MP is invoked. MP first determines the minimum and maximum number of segments (*min_seg* and *max_seg*). Feasible partitions with 3, 4, and 5 segments can be generated for the design. Partitions with 1 and 2 segments are not feasible because no package at level-1 has sufficient area or switching capacity. After determining the minimum and maximum number of segments in feasible partitions, MP invokes MFM to generate partitions with number of segments varying from min_seg to max_seg. MFM calls a k-way Fiduccia-Mattheyses based partitioning algorithm (KWAY– Algorithm 4.6) to generate partitions. MFM keeps track of the best cost partition and returns a constraint satisfying partition, if found, or the best cost partition.

When HPP starts the process of hierarchical partitioning and package design (entering the *while* loop in Algorithm 4.2), it invokes MP with the input graph (in the case of multicomponent synthesis, a process graph; alternately, an RTL netlist), a set of packages available at level-1, a cost constraint ($ 4550), an area constraint (60 sq mm), and a switching activity constraint (5000). Figure 5 illustrates this state and the level-1 partition. Partition segments are marked by dashed lines and the packages partition segments are mapped onto are indicated in text within the segments. A three segment partition with cost $ 3600, area 46 sq mm, and switching activity 1750 is generated. This partition satisfies area, switching activity, and cost constraints and thus MP returns a SUCC status. HPP then uses the hierarchical netlist manager (HN) to read the generated partition and construct a netlist for partitioning at level-2.

Following level-1 partitioning, *Set_Constraint* is invoked to set the cost constraint for the level-2 partition. HPP then invokes MP with the new netlist (generated from the level-1 partition), the set of packages available at level-2, a cost constraint ($ 1200), and area and switching activity constraints (60 sq mm, 5000). Figure 6 illustrates the level-2 partition. A three segment partition with cost $ 1500, area 52 sq mm, and switching activity 1750 is generated. This partition satisfies the area and switching activity constraints, but violates

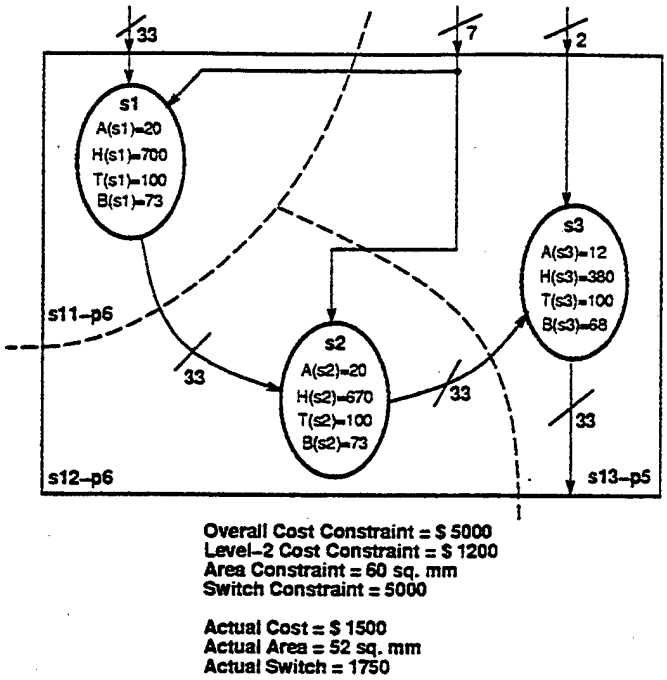


Figure 6: **Level-2 Partition — First Pass**

the cost constraint, thus MP returns a BEST status. HPP now back-tracks to level-1 and starts the second *pass* (a new pass starts every time we back-track to level-1).

On the back-track HPP tightens the cost constraint using Set_Constraint. Assuming a cost overrun factor (COF) of 1, the new cost constraint for the level-1 partition is $ 3300 (as computed by Set_Constraint). HPP re-invokes MP on the RTL netlist (area and switching activity constraints stay the same and the set of packages available at level-1 stays the same). Figure 7 shows the new level-1 partition. A four segment partition with cost $ 3300, area 48 sq mm, and switching activity 1750 is generated. This partition satisfies area, switching activity, and cost constraints and MP returns a SUCC status. HPP then generates a new netlist using HN for level-2.

For the level-2 partition HPP invokes MP with the new netlist and a cost constraint of $ 1500. Figure 8 shows the second pass level-2 partition. A three segment partition with cost $ 1500, area 52 sq mm, and switching activity 1750 is generated. MP returns a SUCC status as area, switching activity, and cost constraints are satisfied. HPP uses HN to generate a netlist for level-3.

A cost constraint of $ 200, an area constraint of 60 sq mm, a switching activity constraint of 1750, and a pin constraint of 75 are considered for the level-3 partition. Figure 9 shows the second pass level-3 partition. A one segment partition with cost $ 400, area 60 sq mm, and switching activity 1750 is generated. MP returns a BEST status as the cost constraint is not satisfied. HPP now back-tracks to level-2.

At level-2 HPP invokes MP with a cost constraint of $ 1300 (as determined by Set_Constraint). Figure 10 shows the second pass level-2 partition on a back-track. A three segment partition with cost $ 1500 is generated. MP returns a BEST status as the cost constraint is not satisfied. HPP now back-tracks to level-1 and begins the third pass.

The third complete pass begins at level-1 with a cost constraint of $ 3100. Figure 11 shows the third pass level-1 partition. A five segment partition with cost $ 3000, area 50 sq mm, and switching activity 1750 is generated. MP returns a SUCC and HPP proceeds to level-2.

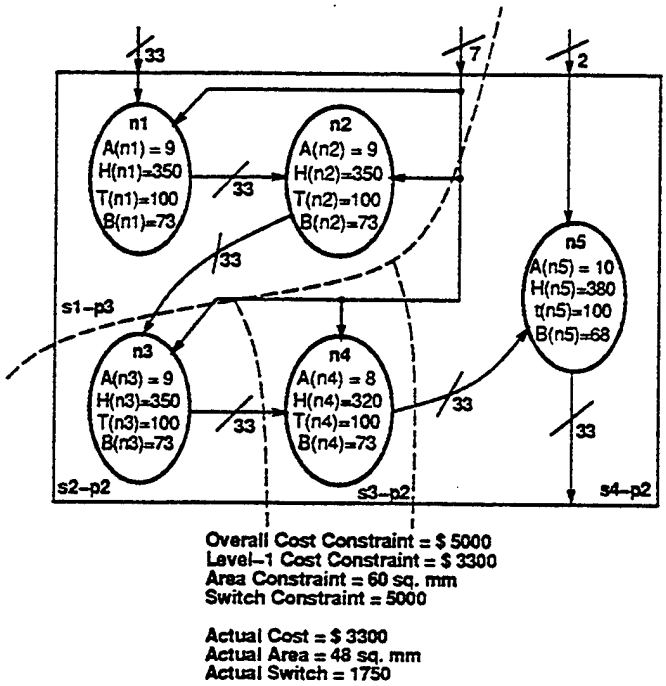


Figure 7: Level-1 Partition — Second Pass (Back-track)

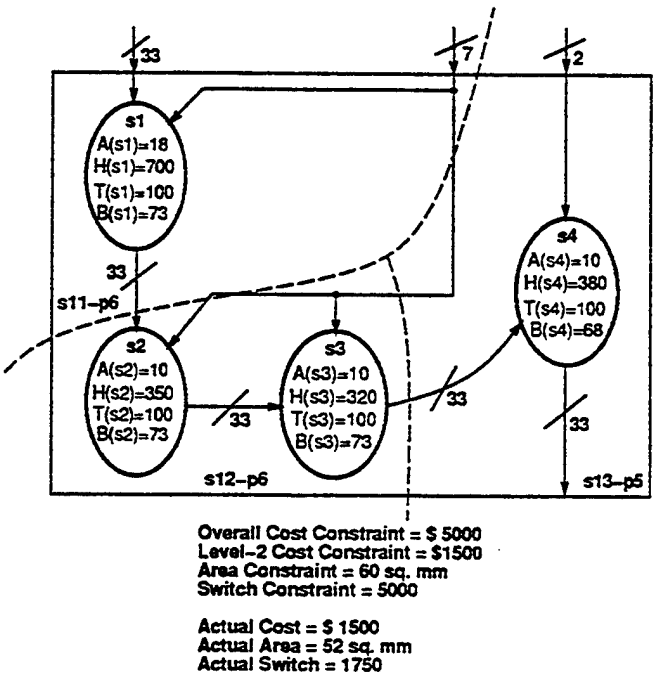


Figure 8: Level-2 Partition — Second Pass

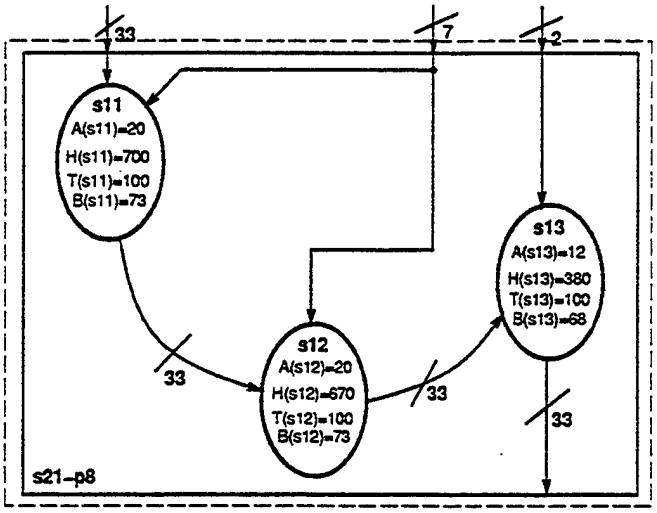


Figure 9: Level-3 Partition — Second Pass

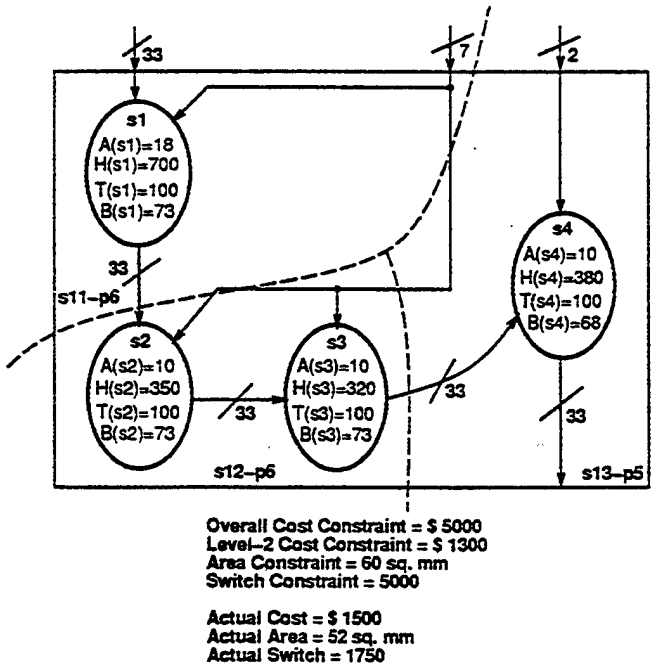


Figure 10: Level-2 Partition — Second Pass (Back-track)

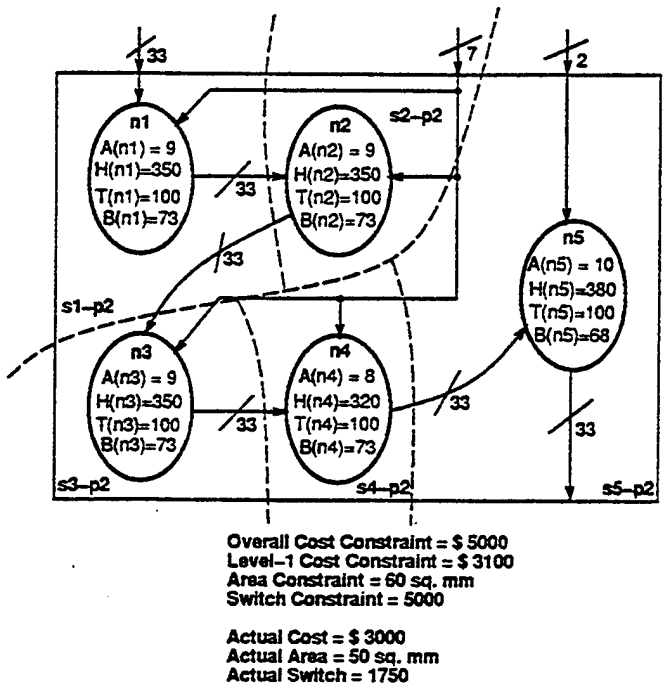


Figure 11: **Level-1 Partition — Third Pass**

HPP invokes MP with a cost constraint of $ 1600 for the third pass level-2 partition. Figure 12 shows the third pass level-2 partition. A three segment partition with cost $ 1500, area 52 sq mm, and switching activity 1750 is generated. MP returns a SUCC and HPP proceeds to level-3.

At level-3 MP is invoked with a cost constraint of $ 500. Figure 13 shows the third pass level-3 partition. A one segment partition with cost $ 400, area 60 sq mm, and switching activity 1750 is generated. MP returns a SUCC. This exhausts the package hierarchy, since there is no level-4 in the package library.

At this point HPP terminates the hierarchical partitioning process and returns the hierarchical design along with the generated package hierarchy (Figures 11,12, and 13). The input RTL design has been successfully mapped onto a hierarchy of packages and a constraint satisfying solution has been found. The overall cost constraint of $ 5000 on the design has been satisfied by finding a solution with cost $ 4900. At each level in the package hierarchy, partition segments have been mapped onto available packages making sure that capacity constraints of the packages are satisfied.

Discussion: In the above example, one of the cases we did not see in HPP is when a FAIL status is returned by MP. A FAIL status indicates that no valid partition for the design exists at this level (i.e., for all feasible partitions at least one of the partition segments could not be mapped onto a package at this level in the package hierarchy). At this point HPP checks if the status of the previous level's partition was a SUCC and, if it is, HPP back-tracks. SUCC at the previous level indicates that there is room for improvement at the previous level and hence the possibility of a valid solution at this level (as a result of improvement at the previous level). If the status of the previous level's partition is BEST, there is no room for improvement at the previous level and HPP terminates reporting failure to find a solution.

Another case we did not observe is what happens when a BEST is returned (by MP) at level-1. When a BEST is also returned at level-2, HPP continues with the partitioning process up the hierarchy. No back-track is attempted because a status of BEST at level-1 indicates that the partition returned is the best cost solution found and cannot be improved. If a SUCC is returned at level-2, HPP could potentially

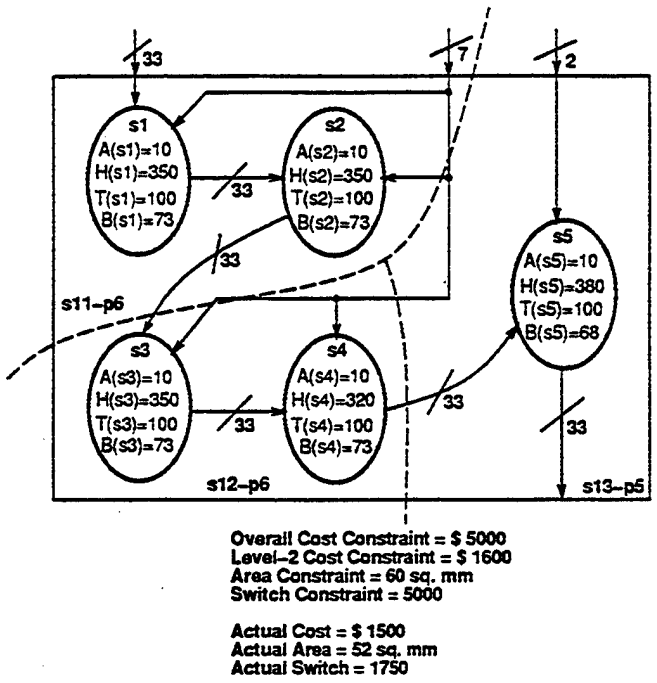


Figure 12: **Level-2 Partition — Third Pass**

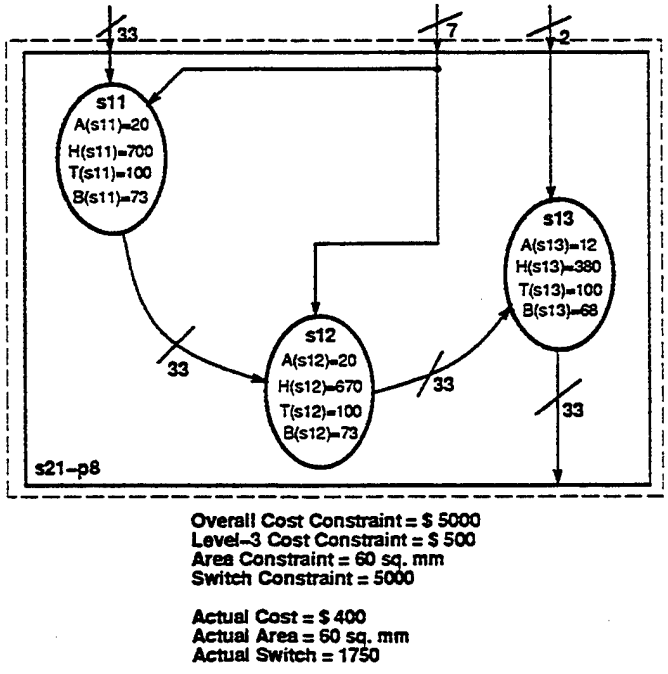


Figure 13: **Level-3 Partition — Third Pass**

back-track to level-2 if level-3 returns a BEST or FAIL. If a FAIL is returned at level-2, HPP terminates the partitioning reporting failure to find a solution. However, in all cases, there is a possibility that an inferior solution at level-1 (a solution other than the best cost solution) could lead to a better overall solution. But due to the nature of costs in VLSI packaging, the highest costs are incurred primarily at level-1 and for some advanced high performance packaging technology such as MCMs at level-2, it is very unlikely that an inferior solution at level-1 could lead to an overall better solution.

The user controls the amount of back-tracking by setting *MaxBtk*. Another way the user can control the amount of back-tracking is by setting initial cost constraint as *zero*. When the initial cost constraint is zero, MP is constrained to always look for the best cost solution (status BEST) at all levels in the package hierarchy. Typically, we find that the solution converges very quickly and we only back-track 2-3 times (see Section 5).

5 Results

We present results for a number of examples to demonstrate the validity of our approach for multicomponent synthesis and hierarchical package design. Details of the package library are shown in Table 2. Data about area, pin, switching activity, and clock speed constraints supported by each package and package cost are presented. We briefly describe the example behavioral specifications. Table 3 presents some details on the number of lines of code and number of processes for each of our examples.

Move Machine: The Move Machine was suggested by Ivan Sutherland based on the observation that conventional processing units spend much of their time moving arguments from memory to CPU and moving results from CPU to memory. The instruction set of the Move Machine merely controls instruction and data flow; it does not compute any data values. Instead, certain memory locations are (assumed to be) connected to external computational units which perform the actual computations. Paul Drongowski provided an ISPS description of a Move Machine in [5]. The VHDL for this example was written by Jay Roy [48, 41]. The description consists of three VHDL processes (fetch, decode and execute) and several internal variables, signals and 'wait' statements.

Fifo: A producer-consumer problem description written using three communicating processes (PRODUCER, CONSUMER, and FIFO). It has five input ports (enqueue, dequeue, a, b, and c) and four output ports (data_ready, z, overflow, and underflow). All data signals and ports (a, b, c, and z) are four bits wide, and all controls (enqueue, dequeue, data_ready, overflow, and underflow) are one bit signals. When *enqueue* goes high, values in ports *a* and *b* are added and stored in the queue. When *dequeue* goes high, value in port *c* is subtracted from the topmost element in the queue and the result is output to *z*. The queue has a depth of 10. If more than 10 values are stored in the queue, *overflow* goes high. Similarly, an attempt to read a value from an empty queue results in *underflow* going high.

Shuffle: The Shuffle is a high speed reconfigurable 32 bit shuffle-exchange network for parallel signal processing. The Shuffle exchange is a commercial product that Texas Instruments, Inc. (TI) used to manufacture (TI part SN74AS8839) [33]. The shuffle-exchange network has a four level architecture that supports five types of multiplexed data permutations: (1) forward shuffle; (2) inverse shuffle; (3) upper broadcast; (4) lower broadcast; and (5) bit exchange. A seven bit control word determines the type of permutation. Additional control is provided with a two bit output selector which determines if the output should be composed of: (1) all 1's; (2) the lower 16 bits – result of 4-level shuffle and upper 16 bits – all 1's; (3) the lower 16 bits – all 1's and upper 16 bits – result of 4-level shuffle, and; (4) all 32 bits – result of 4-level shuffle. The shuffle exchange is modeled as a five process description — each of the four levels of shuffle and the output control are modeled as separate processes.

Level	Name	Area (sq. mm)	Pins	Node Switches	Speed (ns)	Cost ($)
1	Tiny1	5	40	50000	50	400
1	Tiny2	5	40	60000	50	500
1	Tiny3	8	40	80000	50	600
1	Tiny4	12	40	120000	50	700
1	Small1	15	40	150000	50	800
1	Small2	18	40	200000	50	900
1	Small3	20	40	200000	50	1000
1	PGA-1	12	84	200000	50	1200
1	PGA-2	15	84	300000	50	1300
1	PGA-3	18	84	400000	50	1400
1	PGA-4	20	84	500000	50	1500
1	PGA-5	20	84	800000	50	1600
1	PGA-6	20	169	1000000	50	1800
2	Pl-1	6	40	50000	50	250
2	Pl-2	6	40	60000	50	300
2	Pl-3	8	40	80000	50	350
2	Pl-4	12	40	120000	50	400
2	Pl-5	15	40	150000	50	450
2	Cer-1	15	40	200000	50	500
2	Cer-2	18	40	250000	50	550
2	Cer-3	20	40	300000	50	600
2	PGA-1C	12	84	220000	50	800
2	PGA-2C	15	84	320000	50	900
2	PGA-3C	18	84	450000	50	1000
2	PGA-4C	20	84	850000	50	1200
2	PGA-5C	20	169	1000000	50	1500
2	MCM-1	200	169	1000000	75	10000
2	MCM-2	300	169	2000000	75	15000
2	MCM-3	400	169	3000000	75	20000
3	Board-1	300	80	2000000	100	300
3	Board-2	400	80	3000000	100	400
3	Board-3	500	128	4000000	100	500
3	Board-4	600	128	5000000	100	600
3	Board-5	800	128	8000000	100	800
3	Board-6	1000	128	12000000	100	1200

Table 2: **Package Alternatives**

dyn1–dyn10: *dyn* is a five process description that monitors and maintains the dynamic length and maximum length to which a queue in a producer-consumer problem grows. *enqueue* and *dequeue* are used to trigger computation of length and max_length of the queue. *dyn* uses four processes to check for settling of values on enqueue, dequeue, length, and max_length. The fifth process uses a procedure to compute length of the queue depending on enqueue or dequeue and then computes max_length. dyn1–dyn10 are generated by making multiple instantiations (1–10) of the basic five process description of dyn.

alu1–alu5: *alu* is a nine process description of an arithmetic and logic unit (ALU). Eight processes carry out arithmetic and logical operations on a pair of 4 bit inputs. The ninth process uses a 3 bit function select to determine the appropriate function (which arithmetic or logical operation) result to be output. alu1–alu5 are generated by making multiple instantiations (1–5) of the basic nine process description of alu.

We first present results for hierarchical RTL partitioning and multicomponent synthesis and hierarchical package design separately, and then compare the results of the two approaches. Switching activity constraints are not considered in hierarchical RTL partitioning and package design.

Example	Num Lines (VHDL)	Num Proc
Mv Mc	75	3
Fifo	65	3
Shuffle	472	5
dyn1	132	5
dyn2	254	10
dyn3	376	15
dyn4	498	20
dyn5	620	25
dyn6	742	30
dyn7	864	35
dyn8	986	40
dyn9	1108	45
dyn10	1230	50
alu1	100	9
alu2	188	18
alu3	276	27
alu4	364	36
alu5	452	45

Table 3: **Design Data for Examples**

5.1 Hierarchical RTL Partitioning

Table 4 presents experimental results for the hierarchical RTL partitioning approach for the above examples. Number of RTL components in the netlist, mapping of partition segments to packages from the package library, cost of the hierarchical partition (cost of packages partition segments are mapped onto) and cost constraint, and execution time for the designs are presented.

We did not run the *alu* or *dyn* examples with more instantiations (larger example sizes) because execution times for RTL netlists of the larger examples shown in Table 4 are of the order of 30 hours. These examples show a very quick rise in execution times with increase in design sizes (in terms of RTL components in the

Example	No. of Comps	Segments and Mapping (s_i–p_i) Level-1	Level-2	Level-3	Cost/Constraint ($)	Execution Time
Mv Mc	53	s_1–Tiny1 s_2–PGA-6	s_{11}–Pl-1 s_{12}–PGA-5C	s_{21}–Board-1	4250/5000	13.2 s
alu1	65	s_1–Small3	s_{11}–Cer-3	s_{21}–Board-1	1900/2500	6.5 s
Fifo	76	s_1–Small2	s_{11}–Cer-2	s_{21}–Board-1	1750/3000	6.4 s
dyn1	128	s_1–Small1	s_{11}–Pl-5	s_{21}–Board-1	1550/2000	11.9 s
alu2	123	s_1–PGA-3 s_2–PGA-4	s_{11}–PGA-3C s_{12}–PGA-4C	s_{21}–Board-1	5400/5000	49 min 36 s
alu3	161	s_1–PGA-6 s_2–PGA-6 s_3–PGA-6 s_4–Tiny1	s_{11}–PGA-5C s_{12}–PGA-5C s_{13}–PGA-5C s_{14}–Pl-1	s_{21}–Board-1	10850/8000	1 hr 44 min
dyn2	234	s_1–Tiny1 s_2–Tiny1 s_3–PGA-4 s_4–PGA-1	s_{11}–PGA-4C s_{12}–PGA-1C s_{13}–Pl-4	s_{21}–Board-1	6200/3200	1 hr 49 min
alu4	205	23 segments	s_{11}–MCM-3	s_{21}–Board-2	53600/15000	30 hr 31 min
dyn3	334	21 segments	s_{11}–MCM-3	s_{21}–Board-2	53000/3300	31 hr 28 min

Table 4: **Hierarchical Partitioning Results**

Note: s–p denotes the mapping of segment s onto package p from the package library.

design). We did not fully observe the effect of back-tracking on these examples because of the rapidity with which the execution times increased.

5.2 Multicomponent Synthesis and Hierarchical Package Design

Tables 5, 6, and 7 present results of multicomponent synthesis and hierarchical package design for the design examples in Table 3 with the package library shown in Table 2. For each example Table 5 presents: (1) number of processes; (2) hierarchical partition segments mapped onto packages from the package library (at level-1, partitioning of processes (synthesized into equivalent RTL designs) into partition segments); (3) actual number of back-tracks by the hierarchical partitioning and package design algorithm and the limit on number of back-tracks (BTK); (4) actual cost of the design and the cost constraint; and (5) execution time. With a larger number of processes it is difficult to present assignment of processes to partition segments. Table 6 presents the number of processes on each level-1 partition (instead of presenting individual partitions). With an even larger number of processes, it is difficult to present even details of level-2 partition segments. Thus, Table 7 presents the following data for all designs in Table 3: (1) number of processes; (2) number of back-tracks/BTK; (3) actual cost/constraint; and (4) execution time.

We have presented results on multicomponent synthesis and hierarchical package design and hierarchical RTL partitioning and package design. All these results establish and reinforce the validity of our approach. An interesting observation that vindicates our choice of the back-tracking algorithm is that in all our examples the most times we ever back-track is three, in the case of the *alu4* example (Table 7). This is because the algorithm back-tracks only if it can potentially find a solution with better cost and, also, the algorithm focuses in on a solution very rapidly.

Example	No. of Procs	Segments and Mapping (s_i–p_i) Level-3	Level-2	Level-1	Num BkTrk/ BTK	Cost/ Constraint ($)	Exec Time (s)
Mv Mc	3	s_{21}–Board-1	s_{11}–PGA-5C	s_1–PGA-6 EXE	1/10	5600/5000	6
			s_{12}–PGA-1C	s_2–PGA-1 FET, DEC			
Fifo	3	s_{21}–Board-1	s_{11}–Pl-5	s_1–Small1 FIFO PRODUCER CONSUMER	0/10	1550/3000	2.7
Shuffle	5	s_{21}–Board-2	s_{11}–PGA-4C	s_1–PGA-4 shuffle-1	0/10	13900/12000	59.8
			s_{12}–PGA-4C	s_2–PGA-4 shuffle-2			
			s_{13}–PGA-4C	s_3–PGA-4 shuffle-3			
			s_{14}–PGA-4C	s_4–PGA-4 shuffle-4			
			s_{15}–PGA-4C	s_5–PGA-4 output			
dyn1	5	s_{21}–Board-1	s_{11}–Cer-3	s_1–Small3 s1_p_1,s1_p_pt s1_p_sl,s1_p_2 s1_p_st	1/10	1900/2000	3.6
alu1	9	s_{21}–Board-1	s_{11}–Cer-2	s_1–PGA-1 s1_nbp,s1_nap s1_np,s1_outp	1/10	3100/2500	100.7
				s_2–Tiny1 s1_mp,s1_ap s1_op			
			s_{12}–Pl-1	s_3–Tiny1 s1_dp,s1_sp			
dyn2	10	s_{21}–Board-1	s_{11}–Cer-3	s_1–Small-1 s2_p_sl,s2_p_pt s2_p_2	2/10	3350/3200	212.7
				s_2–Tiny1 s2_p_st,s1_p_st			
			s_{12}–Pl-5	s_3–Small1 s1_p_sl,s1_p_pt s1_p_1,s1_p_2			

Table 5: **Multicomponent Synthesis with Hierarchical Package Design Results**

Note: s–p denotes the mapping of segment s onto package p from the package library. Also, at level-1, mapping of processes to partition segments is presented.

Example	No. of Procs	Segments and Mapping (s_i–p_i) Level-3	Level-2	Level-1	Num BkTrk/ BTK	Cost/ Constraint ($)	Exec Time (s)
dyn3	15	s_{21}–Board-1	s_{11}–Pl-3	s_1–Tiny-3 3 procs	1/10	5000/5000	126.1
			s_{12}–Pl-5	s_2–Small1 4 procs			
			s_{13}–Pl-5	s_3–Small1 4 procs			
			s_{14}–Pl-5	s_4–Small1 4 procs			
alu2	18	s_{21}–Board-1	s_{11}–PGA-3C	s_1–PGA-3 6 procs	1/10	6700/5000	412.8
			s_{12}–Pl-5	s_2–Small1 5 procs			
			s_{13}–PGA-2C	s_3–Tiny1 1 proc			
				s_4–Tiny1 3 procs			
				s_5–Tiny1 2 procs			
			s_{14}–Pl-1	s_6–Tiny1 1 proc			
dyn4	20	s_{21}–Board-1	s_{11}–Pl-5	s_1–Small1 5 procs	0/10	6350/8000	229.3
			s_{12}–Pl-1	s_2–Tiny1 1 proc			
			s_{13}–Cer-2	s_3–Small2 6 procs			
			s_{14}–Pl-3	s_4–Tiny3 3 procs			
			s_{15}–Pl-4	s_5–Tiny4 4 procs			
			s_{16}–Pl-1	s_6–Tiny1 1 proc			

Table 6: **Multicomponent Synthesis and Package Design Results (Contd ...)**

Note: s–p denotes the mapping of segment s onto package p from the package library. Also, at level-1, number of processes on each partition segment are presented.

Example	No. of Procs	Num BkTrk/ BTK	Cost/Constraint ($)	Exec Time (s)
Mv Mc	3	1/10	5600/5000	6
Fifo	3	0/10	1550/3000	2.7
Shuffle	5	0/10	13900/12000	59.8
dyn1	5	1/10	1900/2000	3.6
alu1	9	1/10	3100/2500	100.7
dyn2	10	2/10	3350/3200	212.7
dyn3	15	1/10	5000/5000	126.1
alu2	18	1/10	6700/5000	412.8
dyn4	20	0/10	6350/8000	229.3
dyn5	25	0/10	8350/8000	349.5
alu3	27	0/10	12700/8000	579
dyn6	30	1/10	9850/9000	1470.7
dyn7	35	2/10	11200/10000	3141
alu4	36	3/10	14100/15000	1549.4
dyn8	40	1/10	11850/12000	1863.5
dyn9	45	1/10	13800/13000	3684.1
alu5	45	2/10	17750/18000	1626.4
dyn10	50	2/10	16850/15000	6452.2

Table 7: **Multicomponent Synthesis and Package Design Results (Contd ...)**

5.3 Multicomponent Synthesis vs Hierarchical RTL Partitioning

Table 8 presents a comparison of multicomponent synthesis and hierarchical package design and hierarchical RTL partitioning. The following information is presented for each example: (1) number of processes in the behavioral description; (2) number of RTL components in a single-chip synthesized design; (3) number of back-tracks/limit on back-tracks, cost of packaging design, and execution time for (a) multicomponent synthesis and (b) RTL partitioning; and (4) dollar cost constraint for the design. For each example, the better dollar cost solution is bold-faced. RTL partitioning yields better designs for smaller examples where the number of synthesized RTL components is relatively small (< 200). For larger examples multicomponent synthesis clearly out-performs RTL partitioning in the quality of solutions. Also, the time taken by RTL partitioning is more than the time taken by multicomponent synthesis by an order of magnitude (two orders or magnitude for larger examples – e.g., alu4, dyn3).

5.4 Functional Validation

We have presented results on the performance of the multicomponent synthesis and hierarchical package design algorithm (HPP— Algorithm 4.2) for multicomponent synthesis with hierarchical package design and hierarchical RTL partitioning and package design for a number of examples. Another important step is to functionally validate the designs produced. The output of hierarchical partitioning and package design comprises: (1) in the case of multicomponent synthesis, a set of behaviors (VHDL) (corresponding to individual register level segments that together constitute the multicomponent design) to be synthesized into equivalent RTL designs using a high level synthesis system such as DSS [40, 41]; alternately, a set of RTL design segments in the case of RTL netlists; (2) a set of structures (VHDL) that realizes the hierarchical

Example	Num Proc	Num RTL Comp	Multicomponent Synthesis			Hierarchical RTL Partitioning			Cost ($) Constr.
			Btk/ BTK	Cost ($)	Exec Time (s)	Btk/ BTK	Cost ($)	Exec Time (s)	
Mv Mc	3	53	1/10	5600	6	0/10	**4250**	13.2	5000
Fifo	3	76	0/10	**1550**	2.7	0/10	1750	6.4	3000
Shuffle	5	379	0/10	**13900**	59.8	-	-	-	12000
dyn1	5	128	1/10	1900	3.6	0/10	**1550**	11.9	2000
alu1	9	65	1/10	3100	100.7	0/10	**1900**	6.5	2500
dyn2	10	234	2/10	**3350**	212.7	0/10	6200	6560	3200
dyn3	15	334	1/10	**5000**	126.1	0/10	53000	113272	5000
alu2	18	123	1/10	6700	412.8	0/10	**5400**	2976	5000
dyn4	20	-	0/10	**6350**	229.3	-	-	-	8000
dyn5	25	-	0/10	**8350**	349.5	-	-	-	8000
alu3	27	161	0/10	12700	579	0/10	**10850**	6251	8000
dyn6	30	-	1/10	**9850**	1470.7	-	-	-	9000
dyn7	35	-	2/10	**11200**	3141	-	-	-	10000
alu4	36	205	3/10	**14100**	1549.4	0/10	53600	109850	15000
dyn8	40	-	1/10	**11850**	1863.5	-	-	-	12000
dyn9	45	-	1/10	**13800**	3684.1	-	-	-	13000
alu5	45	-	2/10	**17750**	1626.4	-	-	-	18000
dyn10	50	-	2/10	**16850**	6452.2	-	-	-	15000

Table 8: **Multicomponent Synthesis vs Hierarchical RTL Partitioning**

multicomponent design; and (3) a binding of behaviors (RTL segments) and structures to appropriate packages from the package library at each level of the package and design hierarchy. From the viewpoint of functional validation (1) and (2) are of importance. The functional validation approach consists of: (1) synthesizing register level designs from the behavior segments using a high level synthesis system such as DSS(in the case of multicomponent synthesis); and (2) simulating the multicomponent design in VHDL using the same characteristic set of test vectors used for validating the behavioral specification (see Section 3 — *profiling stimuli*). We have functionally validated the Move Machine, Fifo, and Shuffle examples by simulating the output multicomponent designs in VHDL (the other examples — alu1–alu10, dyn1–dyn10 — are synthetic and are used for illustrating the capability of the multicomponent synthesis and hierarchical package design algorithm to handle large designs). In addition to functionally validating these designs at the VHDL level, we have validated the *shuffle exchange* example at the layout level by switch level simulation. We generated the layout of the hierarchical design using the Lager IV silicon compiler [22]. We extracted switch level models from the layouts and simulated the switch level model using IRSIM, a switch level simulator.

6 Conclusions and Discussion

We have presented a generic hierarchical partitioning and package design technique for multichip designs. It takes a generic graph specification (in the case of multicomponent synthesis, a process graph; alternately, an RTL netlist in the case of RTL partitioning), a set of available packaging options, an overall cost constraint on the design and generates a multichip design while simultaneously constructing a physical package hierarchy

for the design. We have demonstrated two applications of this generic technique for multichip design: (1) hierarchical RTL partitioning and (2) multicomponent synthesis with hierarchical package design.

We have presented results for both approaches and also compared the performance of the approaches with respect to the quality of designs produced and execution times for a number of typical design examples. RTL partitioning and package design yields good results for examples where the number of RTL components in the synthesized design are less than 200. But RTL partitioning and package design does not handle thermal (switching activity) constraints on the design and cannot be used for designs where thermal considerations are important. When partitioning at the RTL netlist level, the design architecture is frozen (during high level synthesis). Alternate multichip designs cannot be explored during hierarchical RTL partitioning, whereas multicomponent synthesis explores the design space by considering alternate implementations during high level multicomponent synthesis. Also, thermal profiling of RTL designs is too time consuming (Section 3 and is not viable for large designs. Multicomponent synthesis with hierarchical package design yields better results for the larger examples and also considers switching activity constraints on the design. Also, execution times for multicomponent synthesis are much lower than execution times for RTL partitioning for almost all our examples. Thus, multicomponent synthesis with hierarchical package design is the preferred approach for large designs and high performance packaging technology.

References

[1] M. Beardslee, C. Kring, R. Murgai, H. Savoj, R.K. Brayton, and A.R. Newton, "SLIP: A Software Environment for System Level Interactive Partitioning," *Proc. ICCAD-89,* pp. 280–283, 1989.

[2] R. Burch, F. N. Najm, P. Yang, and T. Trick, "A Monte Carlo Approach for Power Estimation," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* Vol. 1, No. 1, pp. 63–71, Mar. 1993.

[3] R. Camposano and W. Wolf (Eds.), *High-Level VLSI Synthesis,* Kluwer Academic Publishers, Boston, 1991.

[4] M.A. Cirit, "Estimating Dynamic Power Consumption of CMOS Circuits," *Proc. ICCAD-87,* pp. 534–537, Nov. 1987.

[5] P.J. Drongowski, "An Organization-Level Story Board for Agent – A VLSI Designer's Assistant," Internal Report, DSRG, CES Dept, Case Western Reserve University, Jan 1987.

[6] R. Dutta, "Distributed Design-Space Exploration for High-Level Synthesis Systems," *Master's Thesis,* Dept. of Electrical and Computer Engineering, University of Cincinnati, OH, 1991.

[7] R. Dutta, J. Roy, and R. Vemuri, "Distributed Design-Space Exploration for High-Level Synthesis Systems," *Proc. 29th Design Automation Conference,* pp. 644–650, June 1992.

[8] C.M. Fiduccia and R.M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. 19th Design Automation Conference,* pp. 175–181, June 1982.

[9] D.D. Gajski, N.D. Dutt, A.C-H. Wu, and S.Y-L. Lin, *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.

[10] R. Gupta and G. De Micheli, "Partitioning of Functional Models of Synchronous Digital Systems," *Proc. ICCAD-90,* Santa Clara, pp. 216–219, Nov. 1990.

[11] R. Jain, "High-Level Area-Delay Prediction with Application to Behavioral Synthesis," *Ph.D. Dissertation,* Department of Electrical Engineering, University of Southern California, July 1989.

[12] R. Jain, M.J. Mlinar, and A.C. Parker, "Area-Time Model for Synthesis of Non-Pipelined Designs," *Proc. ICCAD-88,* pp. 48–51, Nov. 1988.

[13] R. Jain, A.C. Parker, and N. Park, "Predicting Area-Time Tradeoffs for Pipelined Designs," *Proc. 29th Design Automation Conference,* pp. 35–41, June 1987.

[14] S.C. Johnson, "Hierarchical Clustering Schemes," *Psychometrika,* Vol. 32, No. 3, Sept. 1967.

[15] S. M. Kang, "Accurate Simulation of Power Dissipation in VLSI Circuits," *IEEE J. of Solid-State Circuits,* Vol. 21, No. 5, pp. 889–891, Oct. 1986.

[16] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal,* pp. 291–307, Feb. 1970.

[17] K. Kucukcakar and A.C. Parker, "CHOP: A Constraint-Driven System-Level Partitioner," *Proc. 28th Design Automation Conference,* pp. 514–519, June 1991.

[18] K. Kucukcakar, "System-Level Synthesis Techniques With Emphasis on Partitioning and Design Planning," *Ph.D. Dissertation,* Dept. of Electrical Engineering-Systems, University of Southern California, CA, Oct. 1991.

[19] N. Kumar, "High Level VLSI Synthesis for Multichip Designs," *Ph.D. Dissertation (Draft),* Dept. of Electrical and Computer Engineering, University of Cincinnati, Cincinnati, OH, October 1994.

[20] N. Kumar, L. Rader, S. Katkoori, and R. Vemuri, "Profile-Driven Behavioral Synthesis for Low Power VLSI Systems," *Technical Report ECE-DDE-94-01,* Dept. of Electrical and Computer Engineering, University of Cincinnati, June 1994.

[21] F.J. Kurdahi, "Area Estimation of VLSI Circuits," *Ph.D. Dissertation,* Dept. of Electrical Engineering, University of Southern California, CA, 1987.

[22] *Lager Tool Set,* University of California, Berkeley, 1991.

[23] E.D. Lagnese and D.E. Thomas, "Architectural Partitioning for System Level Design," *Proc. 26th Design Automation Conference,* pp. 62–67, June 1989.

[24] E.D. Lagnese and D.E. Thomas, "Architectural Partitioning for System Level Synthesis of Integrated Circuits," *IEEE Trans. Computer-Aided Design,* Vol. 10, No. 7, pp. 847–860, July 1991.

[25] M.C. McFarland, "Computer-Aided Partitioning of Behavioral Hardware Descriptions," *Proc. 20th Design Automation Conference,* pp. 472–478, June 1983.

[26] M.C. McFarland, A.C. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," *Proc. 25th Design Automation Conference,* pp. 330–336, June 1988.

[27] M.C. McFarland, A.C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proc. of the IEEE,* Vol. 78, No. 2, pp. 301–318, Feb. 1990.

[28] M.C. McFarland and T.J. Kowalski, "Assisting DAA: The Use of Global Analysis in an Expert System," *Proc. ICCAD-86,* pp. 482–485, Oct. 1986.

[29] M.C. McFarland and T.J. Kowalski, "Incorporating Bottom-Up Design into Hardware Synthesis," *IEEE Trans. Computer-Aided Design,* Vol. 9, No. 9, pp. 938–950, Sept. 1990.

[30] M.J. Mlinar, "Control Path/Data Path Tradeoffs in VLSI Design," *Ph.D. Dissertation,* Department of Electrical Engineering-Systems, University of Southern California, CA, June 1991.

[31] J. Monteiro, S. Devadas and B. Lin, "A Methodology for Efficient Estimation of Switching Activity in Sequential Logic Circuits," *Proc. 31st Design Automation Conference,* pp. 12–17, June 1994.

[32] F. N. Najm, "Transition Density: A New Measure of Activity in Digital Circuits," *IEEE Trans. Computer-Aided Design,* Vol. 12, No. 2, pp. 310–323, Feb. 1993.

[33] J. Niehaus and B. Fleck, "Novel IC shuffles parallel-processing data," *Electronic Products,* pp. 42–50, Aug. 1986.

[34] John Ousterhout, "Using Crystal for Timing Analysis," Electrical Engineering and Computer Sciences, University of California at Berkeley, 1987.

[35] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," *Proc. 24th Design Automation Conference,* pp. 195–202, June 1987.

[36] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Trans. Computer-Aided Design,* Vol. 8, No. 6, pp. 661–679, June 1989.

[37] P.G. Paulin and J.P. Knight, "Algorithms for High-Level Synthesis," *IEEE Design & Test of Computers,* pp. 18–31, Dec. 1989.

[38] T.S. Payne and W.M. vanCleemput, "Automated Partitioning of Hierarchically Specified Digital Systems," *Proc. 19th Design Automation Conference,* pp. 182–192, 1982.

[39] M.L. Resnick, "SPARTA: A System Partitioning Aid," *IEEE Trans. Computer-Aided Design,* Vol. 5, No. 4, pp. 490–498, Oct. 1986.

[40] J. Roy, N. Kumar, R. Dutta, and R. Vemuri, "DSS: A Distributed High-Level Synthesis System," *IEEE Design & Test of Computers,* pp. 18–32, June 1992.

[41] J. Roy, "Parallel Algorithms for High-Level Synthesis," *Ph.D. Dissertation,* Dept. of Electrical and Computer Engineering, University of Cincinnati, OH, Feb. 1993.

[42] K. Roy and S. Prasad, "Circuit Activity Based Logic Synthesis for Low Power Reliable Operations," *IEEE Trans. Very Large Scale Integration (VLSI) Systems,* Vol. 1, No. 4, pp. 503–513, Dec. 1993.

[43] Y. Saab and V. Rao, "An Evolution-Based Approach to Partitioning ASIC Systems," *Proc. 26th ACM/IEEE Design Automation Conference,* pp. 767–770, June 1989.

[44] M. Shih, E.S. Kuh, and R-S. Tsay, "Performance-Driven System Partitioning on Multi-Chip Modules," *Proc. 29th Design Automation Conference,* pp. 53–56, June 1992.

[45] C-Y Tsui, M. Pedram, A. M. Despain, "Exact and Approximate Methods for Calculating Signal and Transition Probabilities in FSMs," *Proc. 31st Design Automation Conference,* pp. 18–23, June 1994.

[46] F. Vahid and D.D. Gajski, "Specification Partitioning for System Design," *Proc. 29th Design Automation Conference,* pp. 219–224, June 1992.

[47] R. Vemuri, "Genetic Algorithms for Partitioning, Placement, and Layer Assignment for Multichip Modules," *Ph.D. Dissertation,* Dept. of Electrical and Computer Engineering, University of Cincinnati, OH, July 1994.

[48] R. Vemuri, J. Roy, P. Mamtora, and N. Kumar, "Benchmarks for High Level Synthesis," *Technical Report TM-DDE-91-11,* Dept. of Electrical and Computer Engineering, University of Cincinnati, June 1991, Revised November 1991.

[49] R. Vemuri et al, "An Integrated Multicomponent Synthesis Environment for Multichip Modules," *Computer,* pp. 62–74, April 1993.

[50] R. Vemuri et al, "Experiences in Functional Validation of a High Level Synthesis System," *Proc. 30th Design Automation Conference,* pp. 194–201, June 1993.

[51] R. Vemuri and R. Vemuri, "Partitioning for Multichip Modules," *Electronics Letters,* Vol. 30, No. 16, pp. 1270–1272, Aug. 1994.

[52] R. Vutukuru, "Test bench compilation for synthesized multicomponent designs," *Master's Thesis,* University of Cincinnati, 1992.

[53] R.A. Walker and D.E. Thomas, "Behavioral Transformation for Algorithmic Level IC Design," *IEEE Trans. Computer-Aided Design,* Vol. 8, No. 10, pp. 1115–1128, Oct. 1989.

APPENDIX H:

Performance Modeling and Tradeoff Analysis During Rapid Prototyping[1]

Jeffrey Walrath, Karam S. Chatha, Ranga Vemuri,
Naren Narasimhan and Vinoo Srinivasan
Laboratory for Digital Design Environments
Department of ECE & CS, University of Cincinnati
Cincinnati, OH 45221–0030
Ph: 513-556-4784; Email: ranga.vemuri@uc.edu

Abstract

Tradeoff analysis is a central aspect of any design process. Languages and tools to support performance modeling and evaluation are necessary to facilitate rapid prototyping of designs. A performance modeling and tradeoff analysis environment reduces the overall design time of both the prototype and the final product, by helping designers in determining which parameters of a design are critical for meeting a set of desired performance goals. This paper describes a case study in performance modeling using a language called PDL (Performance Modeling Language). The PDL system supports tradeoff analysis and performance visualization. This paper also addresses some of the key issues for successful tradeoff analysis during rapid prototyping and explain how many features of PDL make it a suitable choice for this purpose.

1 Introduction

During any design process, many decisions are made which affect the overall performance of the design. Many such decisions result from detailed tradeoff analysis among several related attributes of the design. For example, choice of the input clock frequency depends partly upon the desired upper bound on power consumption and the desired lower bound on the throughput rate. Decisions such as these are made at various levels of the design process from specification to implementation. In order to make effective design choices, the design environment and supporting tools must be well suited for performing *tradeoff analysis* throughout the design process, at multiple levels of abstraction.

[1]This work was partially supported by the ARPA RASSP program and monitored by the Wright Lab, US-AF under contract number F33615-93-C-1316 and by the Semiconductor Research Corporation under contract number DJ-293.

Performance modeling and tradeoff analysis involves developing a model of the design at some level of abstraction, behavioral, macro-level, register-transfer level etc. During model evaluation, various parameters of the model are altered and their effect on other parameters is observed. Based on this data further design choices are made. The model is modified accordingly, and the process of performance evaluation and tradeoff analysis repeated. This process continues until an acceptable design is reached which meets all the required performance goals.

For effective tradeoff analysis to occur during rapid prototyping, the design environment should have the following features: (1) Modeling at any level of abstraction must be supported since tradeoff analysis occurs at various levels during the design process; (2) The modeling environment must lend itself to reusability. Reuse of models is very critical because it reduces the time spent in writing a model for each new version of the design. (3) The performance evaluation engine should be flexible enough to partially evaluate a model when variations in some parameters are unknown. This facilitates incremental analysis of the model; (4) The modeling environment must be easy to use so that the development and analysis of performance models can be done quickly and efficiently.

We have developed a modeling and tradeoff analysis environment, the PDL System, which meets all these criterion and is well suited for use during rapid prototyping [1]. A PDL program declares design objects (various kinds of modules, nets, and ports) that can appear in a design. In addition, the containment and connectivity relationships among the various kinds of objects can also be declared in the PDL program. When a specific design in the form of a PDL net-list file is compiled with a PDL program, the compiler will be able to determine if the net-list contains objects of the kind declared in the PDL program and whether the net-list structure conforms to the object composition relationships (containment and connectivity) which were declared in the PDL program.

In addition, a PDL program declares various types of attributes and attaches them to the design objects. Attribute evaluation rules can also be specified and attached to the design objects. A PDL program does not specify any order among these rules; they are viewed as mathematical equations. Given a design net-list that conforms to the PDL program, the PDL compiler generates a global attribute dependency graph and automatically infers a complete evaluation order among the various attribute evaluation rules. An executable performance model containing the proper evaluation sequence for all the evaluation rules is generated.

Figure 1 illustrates the PDL System and design process for generating and evaluating performance models. Once a model has been compiled, the evaluator can be configured to evaluate and collect data in several different ways. In the simplest case, a model can be completely evaluated with a complete set of input data. A configuration can be specified that allows for the collection of data for graphical analysis or tabulation. This includes allowing the specification of ranges of values for particular input attributes. Incremental evaluation is also possible with a performance model (the feedback loop in the figure). During evaluation, only some of the input data is supplied with the result being another performance model. Further evaluation on this model can be done with more input data specified as necessary.

This paper outlines the features of the PDL System and through a hardware/software co-design example illustrates how the PDL system can be used for effective tradeoff analysis during rapid prototyping. The rest of this paper is organized as follows: Section 2 introduces the hardware/software co-design example to be used as the case study in this paper. Section 3 develops the PDL performance model of the co-design example and, through this example,

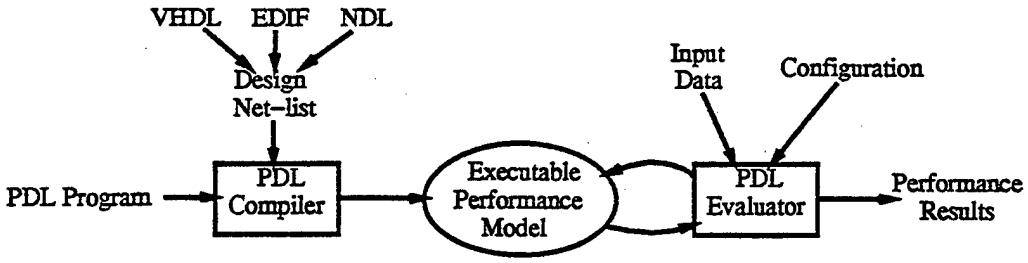


Figure 1: Overview of the PDL System

introduces the PDL language itself. Section 4 describes the performance evaluation and trade-off analysis process using the PDL system. Section 5 discusses the results of this analysis for a specific co-design example: a JPEG-like image compression scheme. Section 6 contains some concluding remarks.

2 Hardware/Software Co-design with Coprocessors

Rapid prototyping for hardware-software co-design of embedded processor and coprocessor system is current research area. The specification for co-design is usually represented as a *task graph* where nodes represent tasks and edges represent communication channels [2]. For hardware-software co-design, the goal is to determine which tasks should be implemented in hardware or software based upon some performance criterion. When the target architecture is an embedded system, several hardware tasks can be implemented as ASICs and all of the software tasks are allocated to execute on an embedded processor. In a coprocessor system, only a single task can be allocated to hardware and all other tasks are allocated to software. Again all software tasks usually run on the same processor. A *coprocessor* is a configurable plug-in board connected to a main processor such as a workstation or a personal computer. Components of the coprocessor board include a programmable chip, interface memory, and a predefined interface protocol to the main computer [3].

While developing a performance model for co-design, several factors shall be considered. From a software perspective, tasks have certain properties that govern their execution sequence. If all tasks are bound to execute in software, then only one task can execute at a time. The next scheduled task can not begin execution until all preceding tasks are finished. Figure 2 is an example of a task graph. Although there are tasks which appear to be independent of each other, an execution order is associated with this task graph since one task can execute at a time. For this example, task 6 can not begin executing until tasks 3,4, and 5 finish. This execution order is referred to as the the *task schedule* for a particular task graph.

Another feature of hardware software co-designs is the inherent parallelism available between the hardware and software. This is achieved by having one hardware task executing in the coprocessor simultaneous with a software task executing in the main processor. Figure 3 shows a simple task graph were hardware software parallelism can be exploited. When none of the tasks are bound to hardware, the only task schedule is task 1 followed by task 2 and so forth. If task 2 where bound to hardware, then the task schedule could be pipelined so that task 2 and 3 operate simultaneously. Pipelining can occur with a data buffer between task 2 and 3.

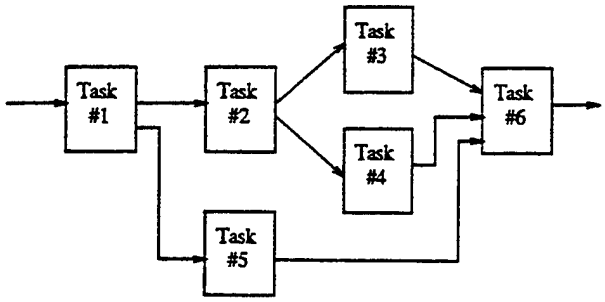
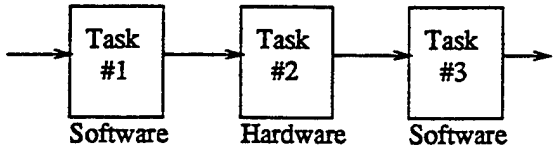


Figure 2: Example Task Graph

After task 2 finishes executing the first time, the data would move to the buffer at the input of task 3. Then, the next time task 2 executed, task 3 would also execute. The buffer is necessary to ensure task 2 can not write to the same memory being read by task 3 during execution.



Time

Task No.	t1	t2	t3	t4	t5	t6	...
1	X		X		X		...
2		X		X		X	...
3				X		X	...

Table: Task Schedule

Figure 3: Exploiting Parallelism in Task Graph

If the target architecture is a coprocessor board, another modeling parameter is the communication time between tasks when one task is in software and the other is in hardware. The computer can not transmit data directly to the coprocessor. Instead, data is transmitted through memory located on the coprocessor board. Because this memory is located on a board which is connected to a slower bus interface, communication time necessary to read and write data from the coprocessor to the computer's main memory is slower than usual memory transfers within the computer. This will have a noticeable impact on estimating execution time of a particular set of bindings. In most co-design problems, the communication between the hardware is such that one task writes to the coprocessor memory prior to execution of the hardware task. Once the hardware task finishes, the next software task reads the results from the coprocessor memory.

The expression for calculating the execution time of a task graph is based on a sum of the execution time for each task. However, with the hardware parallelism that can occur due to pipelining, it is not a simple summation of execution times. In addition, there is a task schedule that has to be specified to start and finish the pipeline. All of these factors must be expressed by the equations for calculating total execution time.

Calculating the execution time for an individual task is given by the equation:

$$ExecutionTime = BindingTime + \sum RdOverHd + \sum WrOverHd \quad (1)$$

$$RdOverHd = NumVariables * ReadTime \quad (2)$$

$$WrOverHd = NumVariables * WriteTime \quad (3)$$

In this equation, *BindingTime* is the execution time for a particular task depending upon a hardware or software binding. As previously mentioned, tasks which must read or write to the coprocessor memory have an associated communication time related to transferring the data. Recall that a single task can have several edges which are input to the task. For each task on the input which transmits data via coprocessor memory, *RdOverHd* will be a non-zero value. If coprocessor memory is not involved, then *RdOverHd* will be zero for that particular input edge. Thus, the execution time includes adding all the time necessary for reading data from the coprocessor memory. A similar addition is used for writing to coprocessor memory for all the outputs and is accounted for by *WrOverHd.*

Calculating total execution time is given by the equation:

$$GlobalTime = \sum_{task1,task2,...} max(task1'ExecutionTime, task2'ExecutionTime, ...) \quad (4)$$

GlobalTime is a sum of the execution times for each task. A particular task is scheduled and the execution time is *ExecutionTime.* Another task is scheduled and its execution time is added to the previous time. This process continues until all tasks have been scheduled with *GlobalTime* accumulating the execution times for each task. Because pipelining allows more than one task to execute at a time, the total global time only increases by the maximum *ExecutionTime* of all tasks which are scheduled to execute simultaneously.

3 Performance Model for Co-designs

In this section, we develop a suitable performance model in PDL for co-design performance estimation. PDL has three basic object types for representing designs: *modules*, *carriers*, and *ports.* A module can be used to represent any type of component typically found in a design. A carrier is commonly used for representing transport components such as connections, wires, buses and communication channels. Ports are objects used primarily for representing the *connectivity* among various design components. [4]

In the co-design example, a task graph represents the overall design and nodes in the task graph represent tasks. Connections between tasks are considered directed edges with no two tasks having more than one directed edge between them. To represent a task graph in a PDL model, the first step is to define the various task graph components with PDL objects. Figure 4 shows the PDL definitions for two ports and a carrier which collectively represent edges in a task graph. Two ports are defined such that there is a unique input and output port which represents a directed edge.

```
port task_out_port
end port;

port task_in_port
end port;
```

```
carrier edge
 ports
  input : task_out_port;
  output : task_in_port;
end carrier;
```

Figure 4: **PDL declaration for representing edge and related ports**

In the carrier declaration edge there is a **ports** section. In PDL, various objects can contain references to other objects; this is known as *containment.* A carrier object may only contain ports. However, a module object may contain references to other modules, carriers, and ports. Containment serves two useful purposes. First, it allows the parent object, in this case the carrier, access to information within any contained object. Secondly, if two different PDL objects, perhaps two modules, contain a reference to the same port, then the two objects are considered connected to each other through that port. Figure 5 shows the PDL definition for a `task` module which represents tasks in the task graph and the `codesign` module which represents the entire task graph.

```
module codesign
 ports
  inputs{} : task_out_port;
  outputs{} : task_in_port;
 carriers
  connections{} : edge;
 modules
  tasks{} : task;
end module;
```

```
module task
 ports
  inputs{} : task_in_port;
  outputs{} : task_out_port;
end module;
```

Figure 5: **PDL declaration for representing task and codesign**

In the `task` module there are containment declarations for `inputs` and `outputs`. Within a task graph, a task may have any number of other task edges as input. Additionally, a task can also have output edges that branch to other tasks. In the declaration, the {} notation is used to denote a set of objects, with a set containing zero or more objects. Thus, for the `task` there will be a set of input and output edges. In a containment declaration, when the {} is not used, this means the reference is to a single object.

Module `codesign` is used to represent the entire task graph. It contains a definition for a set `inputs`. These are all the inputs to the task graph (there may be more than one but is usually the root of the task graph). Another definition declares a set of `outputs`. These are all the outputs from the task graph which are usually the final tasks in the task graph to execute. In addition, there are definitions for `connections` which are all the edges in the task graph and `tasks` which are all the tasks in the graph. The `codesign` module represent all the containment relationships existing in a graph.

Once all the objects representing components in the task graph have been specified, the next step for developing a model is to introduce *attributes* and *attribute evaluation rules* in the

objects. Attributes are parameters that are propagated and computed in the PDL model. An evaluation rule describes how to perform the calculation of an attribute in an object. Figure 6 shows the declarations of the port and carrier objects with all their attributes and evaluation rules.

```
type
 hw_sw_bind : enum {hw,sw};
end type;

port task_out_port
 attributes
  primitive num_var : int;
  t1_bind : hw_sw_bind;
  t2_bind : hw_sw_bind;
  wr_overhd : real;
  rd_overhd : real;
  dynamic done, t2_job : int := 0;
 rules
  wr_overhd =
   wr_comm(t1_bind, t2_bind, num_var);
  rd_overhd =
   rd_comm(t1_bind, t2_bind, num_var);
end port;

port task_in_port
 attributes
  t2_bind : hw_sw_bind;
  rd_overhd : real;
  dynamic done : int := 0;
  dynamic t2_job : int := 0;
end port;

carrier edge
 ports
  input : task_out_port;
  output : task_in_port;
 rules
  input't2_bind = output't2_bind;
  input't2_job = output't2_job;
  output'rd_overhd = input'rd_overhd;
  output'done = input'done;
end carrier;
```

Figure 6: **Attributes for edge carrier and related ports**

Attributes are defined in the **attributes** section of an object. An attribute can be any allowable data type. Some of the types available are integer, real, enumerated type, heterogeneous records, lists, and a variety of combinations of these. An attribute is associated with the object where it was declared and not with the object where the attribute is given a value or referenced. Thus, when an attribute is used in an expression where it is not defined within the object, it is referenced as `object'attribute`. For example, in the edge carrier there is a reference to `input't2_bind`. The port `input` is declared as a contained port and within the port there is an attribute declaration for `t2_bind`.

Along with defining the type, the `attributes` section is used to define an attribute as *primitive* or *non-primitive*. An attribute is non-primitive unless explicitly declared as a primitive. A primitive attribute is an attribute which will not have an evaluation rule for defining how to calculate it. Instead, a primitive attribute will have its value set by the user during the execution of the performance model. For example, in the `task_out_port` port the primitive attribute `num_var` is the number of variables being transmitted from one task to another. This value can not be calculated because it depends on the actual task graph being modeled and is not based on any information within the model. Thus, when the model is executed the user will supply the number of variables being transmitted.

In addition to being primitive, an attribute can also be *dynamic* or *static*. An attribute is considered static unless declared dynamic. During model execution, all static attributes are calculated once. These are attributes which are not based upon some dynamic stream of events, but instead are values which need to be calculated once since they are independent of

other events occuring within the model. Conversely, dynamic attributes are not single values but streams of single values. As a model executes, it may be re-evaluated any number of times. During each re-evaluation cycle, there is a corresponding value for the dynamic attribute. For example, if there are 5 evaluation cycles, then every dynamic attribute will have 5 distinct values. This is similar to simulating the performance model based on a stream of events which occur.

Along with defining attributes and types, evaluation rules also need to be defined for various attributes. It is not necessary that an attribute has an evaluation rule in the same object where it was declared. For instance, in the `task_in_port`, all the attributes are given values by the edge carrier object. Thus, the `task_in_port` port has no evaluation rule for these attributes. This is how information is transmitted among various objects in a PDL model. An attribute is declared for some object, but another object has an evaluation rule for the attribute. Another object can reference the value of the attribute after it has been evaluated. For example, in the edge carrier, the done attribute of output is given an evaluation rule where it is the same as `input'done`. Any other object which would contain the same input port could then read the value of done.

Figure 7 shows the `task` module with all its corresponding attributes and evaluation rules. There are several evaluation rules which transfer information between the input and output of the task. These attributes are used for defining when a task has been scheduled and to determine the bindings of connected tasks. Recall that if a task is in hardware there is a communication overhead which must be calculated. Attribute `comm_overhd` will be 0.0 if the task is not bound to hardware otherwise it will be a total of `rd_overhd` and `wr_overhd` times which were calculated in the port using equations 2 and 3. The `exec_time` attribute is either the hardware or software time for the task, and the `time` attribute is the sum of the execution time and communication time. This is the total execution time for the task when it is scheduled. Because every dynamic attribute is re-evaluated on each evaluation cycle, if a task has not been scheduled it does not add to the total time during that specific evaluation cycle. A new task is scheduled each evaluation cycle.

The last definition is the `codesign` module. Figure 8 shows the definition with all the attributes and evaluation rules. In the `rules` section, there are several evaluation rules which set attributes in the contained object `tasks`. The {} indicates that attribute `num_jobs` in `task` is to be set to the primitive value `num_jobs`. The evaluation rule for `global_time` is similar to equation 4. There is an evaluation rule which sets the `global_time` in each task to the current `global_time`. Thus, after each task is scheduled and calculated, a new `global_time` is determined by taking the maximum of all the times from each task.

All of these evaluation rules have been defined in the PDL model in no particular order. However, there is an inherent order associated that is implied by these rules. If a rule depends upon the value of another rule, then that rule can not be evaluated until the other rule is done first. These evaluation rule dependencies produce a global attribute dependency graph. Figure 9 illustrates just some of the dependencies among the various attributes in the `task` module. Primitive attributes do not depend upon other attributes and are the leaf nodes in the dependency graph (those attributes in the figure with boxes around them). A directed dependency graph may not contain any cycles among the attributes.

```
module task
 ports
  inputs{} : task_in_port;
  outputs{} : task_out_port;
 attributes
  primitive binding : hw_sw_bind;
  primitive hwtime : real;
  primitive swtime : real;
  primitive schd_no : int;
  primitive dynamic schd_task : int := 0;
  dynamic time : real := 0.0;
  dynamic job : int := 0;
  dynamic job_diff : int := 0;
  dynamic done_in : int := 0;
  dynamic done_out : int := 0;
  dynamic exec:int := 0;
  dynamic global_time : real := 0.0;
  num_jobs : int;
  rd_overhd : real;
  wr_overhd : real;
  comm_overhd : real;
  exec_time : real;
 rules
  inputs{}'t2_bind = binding;
  inputs{}'t2_job = curr job;
  outputs{}'t1_bind = binding;
  exec_time = if (binding == hw)
   then hwtime
   else swtime
   endif;
  done_in =
   eval(foreach p in inputs{ curr p'done });
  job_diff = curr job -
   min(foreach p in outputs{ p't2_job });
  rd_overhd =
   sum(foreach p in inputs{ p'rd_overhd });
  wr_overhd =
   sum(foreach p in outputs{ p'wr_overhd });
  exec =
   begin
   temp:int;
   if ((done_in == 1) and (curr job < num_jobs)
     and (job_diff < 1))
    then if (binding == hw)
     then temp:=1;
     else if (schd_no == schd_task)
       then temp:=1;
       else temp:=0;
       endif;
     endif;
   else temp:=0;
   endif;
   return temp;
  end;
  comm_overhd = if (binding == hw)
   then 0.0
   else wr_overhd + rd_overhd
   endif;
  time = if (exec == 1)
   then global_time + exec_time + comm_overhd
   else time
   endif;
  job = if (exec == 1)
   then job + 1
   else job
   endif;
  done_out = if (exec == 1)
   then 1
   else done_out
   endif;
  outputs{}'done = done_out;
end module;
```

Figure 7: **Attributes for the task module**

```
module codesign
 ports
  inputs{} : task_out_port;
  outputs{} : task_in_port;
 carriers
  connections{} : edge;
 modules
  tasks{} : task;
 attributes
  primitive num_jobs : int;
  primitive dynamic schd_task : int := 0;
  dynamic global_time : real := 0.0;
 rules
  tasks{}'num_jobs = num_jobs;
  tasks{}'schd_task = schd_task;
  global_time =
   max(foreach t in tasks{t'time});
  tasks{}'global_time = curr global_time;
end module;
```

Figure 8: **Attributes for the codesign module**

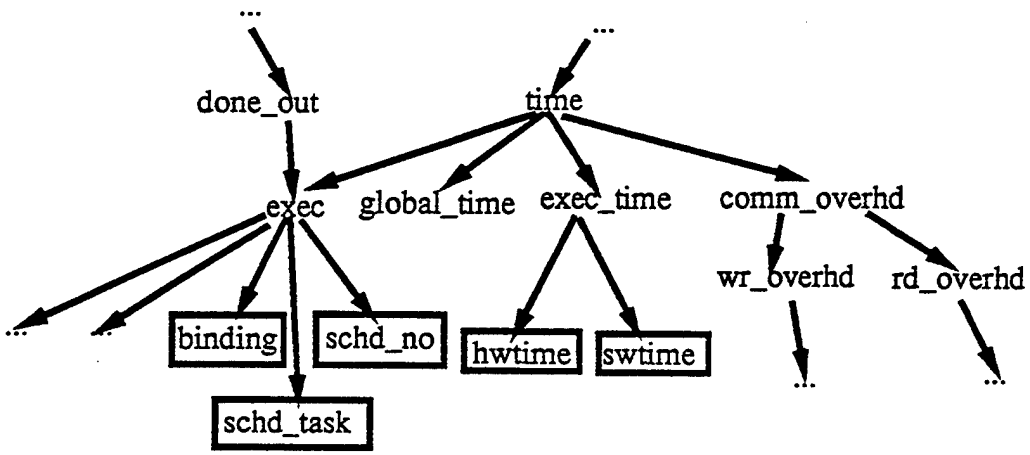


Figure 9: **Example Dependency Graph**

4 Tradeoff Analysis Using the PDL System

Once a PDL model is written, the next step is to compile it. As mentioned previously, a PDL model by itself is not an executable model. An executable model is only generated when the PDL model is coupled with a specific design. This is the role of the compiler. It takes a PDL model and a design (a specific task graph in the case of our example) as input and generates an executable performance model. During the performance model generation, the order for evaluating all the various attributes is determined. The result of compilation is an executable performance model has a correct evaluation order for all expressions. Figure 10 is a detailed overview of the PDL system and the flow of a PDL program and net-list through an analysis cycle.

Once a performance model has been generated, the user executes the model with the PDL *evaluator*. Model evaluation can be done in several ways depending upon the configuration and input to the evaluator tool. A performance model can be completely evaluated when all

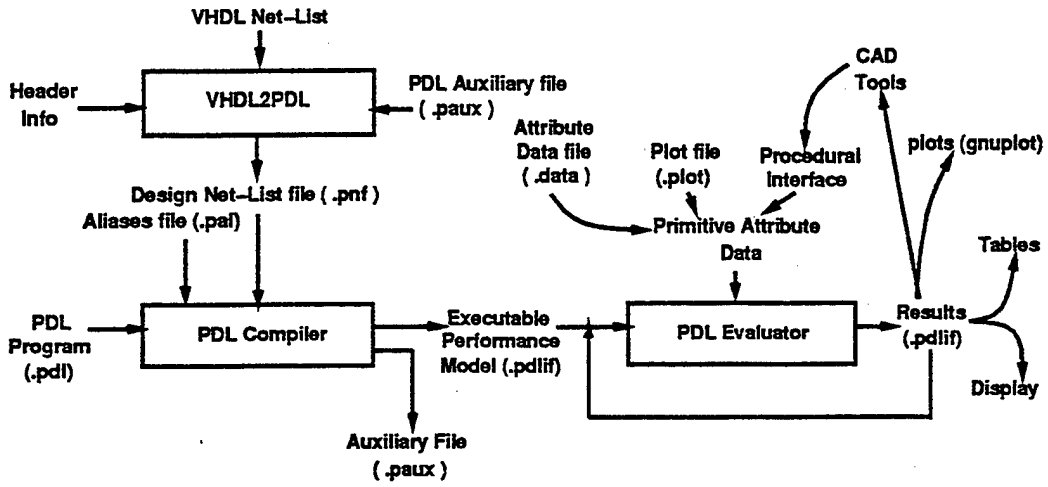


Figure 10: PDL System Overview

the primitive attribute values are supplied; this is known as *full evaluation.* Another option is to evaluate the performance model with only some of the primitive data specified. This is known as *partial evaluation.* In addition, the evaluator can be configured to collect data for analysis based on ranges of values for primitive attributes instead of single values. Finally, the evaluator can be linked into an existing CAD/CAE tool to perform data analysis.

Full evaluation of a model begins with a performance model. All primitive attributes that were defined in the PDL model must be defined by the user in an input data file. When the evaluator is invoked, both the performance model and data file are read, and all expressions are evaluated with the results written to another performance model. Since the model was fully evaluated, all evaluation rules will have been replaced with their corresponding evaluation result. Thus, the resulting performance model will contain nothing but attributes and their evaluated values.

In addition to full evaluation, the user can partially or incrementally evaluate a model. Instead of specifying a complete set of values for all primitive attributes, the user can specify only some of the data for the primitive attributes. When the evaluator is invoked, the performance model and data file are read, all evaluation rules are partially evaluated and a residual performance model is generated. The residual model model will still contain (partially evaluated) evaluation rules for various attributes which have been reduced and simplified with respect to the original expression. The residual model can be further evaluated when more primitive attribute data is available.

There are several cases where partial evaluation can be useful during tradeoff analysis. During analysis, there may be some primitive attributes of interest that need analysis as to their effect on the designs performance. Evaluating a large model several times with primitive attributes which do not change between successive evaluations can be costly. The solution is to partially evaluate the performance model with only those primitive attributes which do not change. All evaluation rules are evaluated and whenever possible reduced to depend only on those primitive attributes which have not been specified. This results in a simpler performance model that can then be used in successive evaluations with the remaining primitive attributes specified.

This results in the elimination of redundant evaluation of unchanging evaluation rules which helps to improve data collection.

In addition to evaluating a model with single data points, the evaluator can be configured to collect data for ranges of primitive values. For example, instead of setting a primitive attribute to one particular value, the user can specify that a primitive attribute can be a range of values. Then during execution, the model is evaluated with the specified attribute set to each value in the range. Any number of primitive attributes can be setup to have ranges of values. In addition, the evaluator can be configured to collect data on any attribute attribute within the model, primitive or non-primitive. Two types of data collection is possible: Data can be collected in a format suitable for two or three dimensional plots, or the evaluator can collect data for any number of attributes and store the results in tabular form.

In the case where a designer may need to collect data in a particular fashion not handled by the evaluator, the evaluator exists as a C run-time library. Contained in the library are several functions which together constitute a procedural interface to the evaluator. The user can use these functions to setup a performance model and collect data in a form suitable for their own needs. Thus, the library can be used to read a performance model, set values for various attributes, evaluate the model, restore the performance model to a previous state, and many other activities.

5 Tradeoff Analysis for a Codesign Example

The co-design model written in PDL and discussed in Section 3 is flexible enough to perform performance modeling for many different types of task graphs. In addition, any number of tasks can be bound to hardware or software. In this section, we discuss results of using this model for a specific codesign example involving a JPEG-like compression/decompression scheme [5, 6]. The target architecture was a coprocessor system. Tradeoff analysis was performed with the PDL model to determine which task to implement in hardware. Figure 11 shows the task graph for the compression part of the JPEG algorithm in terms of objects in the PDL model. Arrows in the figure represent the connectivity among the various PDL objects.

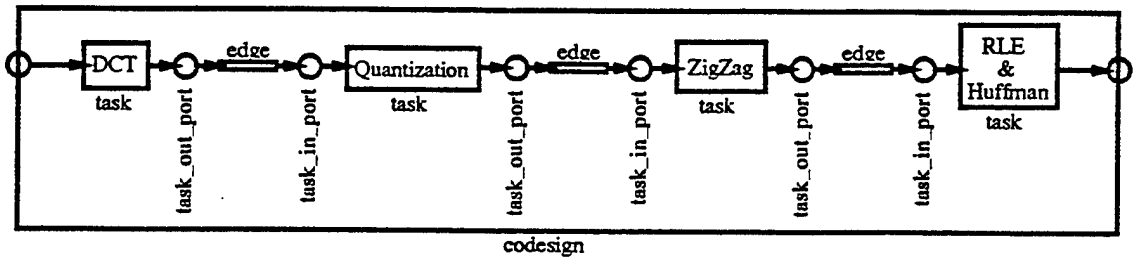


Figure 11: Task Graph for JPEG

First step in performing tradeoff analysis was estimating the hardware and software times for each task. Obtaining software time involved using existing software profiling tools to time each of the tasks in the software version of the JPEG algorithm. In this case, all software times were collected with timing functions on a Pentium system containing a P100 microprocessor,

256 kilobytes of standard cache, and 16 megabytes of main memory. Estimating the hardware execution times was accomplished with a synthesis tool [7] that generated a register transfer level design for each task. In addition the synthesis tool estimated the execution times for each RTL design. Times were estimated for a 2 micron CMOS technology. Table 1 shows estimated hardware and measured software execution times for the various JPEG tasks. These times are for each task performing its respective job on 16 pixels at a time.

Task	Hardware	Software
DCT	8.4 μs	371.3 μs
Quantization	0.6 μs	7.56 μs
ZigZag	0.4 μs	1.63 μs
RLE and Huffman Encoding	884 μs	18.48 μs

Table 1: **Estimated Task Times**

There are several aspects of any task which affect its behavior in hardware or software. Tasks which are very mathematically intensive tend to have better performance in hardware than software because of hardware optimizations made by the the synthesis tool. However, task which contain many control and data flow statements are better suited for software because the synthesized control hardware is far more complex than its software counterpart. Execution times in table 1 illustrate these facts. The DCT (Discrete Cosine Transform) is almost entirely mathematical and as such performs better in hardware than software. However, Huffman encoding is a control oriented algorithm containing very few arithmetic operations.

The next step in the analysis process was to use the PDL model to determine execution times for the task graph with each task bound to hardware. This was done by compiling the PDL program with the design for the JPEG task graph. Four data files were created as input to the evaluator with each file binding a single task to hardware. The model was setup to estimate execution time for an input file that contained 4080 pixels. In addition, all task schedules were defined for pipelining since the PDL model was written to account for it. Table 2 shows the results of evaluating the model with these four data files.

Task in Hardware	Execution Time
DCT	0.234 s
Quantization	1.51 s
ZigZag	1.54 s
RLE and Huffman Encoding	4.07 s

Table 2: **PDL Results for Task Bindings**

Results of table 2 show that the DCT (Discrete Cosine Transform) task was the best choice for implementation on the coprocessor hardware. We did implement the DCT task in a coprocessor system [3] connected to a Pentium based PC. Once complete, actual execution times for compressing images of different sizes were measured. Accordingly, the PDL performance model was evaluated with primitive attributes set for each of the different input images. Table 3 shows the results of the PDL estimations compared with the coprocessor execution times.

File Size (no. of pixels)	PDL Estimated Time (seconds)	Actual Time (seconds)	% Error
18,048	2.23 s	2.11 s	5.7
25,920	3.09 s	3.02 s	2.3
54,896	6.13 s	6.51 s	5.8
69,840	7.43 s	8.11 s	8.3
87,552	9.16 s	10.16 s	9.8

Table 3: **Comparison of Estimated to Actual Execution Times**

6 Conclusion

We have presented a performance modeling and analysis approach for co-designs using the PDL system. In PDL, it is straight-forward to make several enhancements to the codesign model presented in this paper so that it requires less primitive input information or considers more performance parameters than just execution time. For example, the model could determine a task schedule based on the hardware software bindings, more detail could be included as to the target architecture, estimation could be incorporated for cost, hardware area, and so forth. As the design evolves and requires more detailed performance analysis, so too can the PDL model evolve and be refined to a more accurate representation of the system being modeled.

It is important to note that the PDL model is a generic model from which specific, executable performance models can be generated (using the PDL compiler) given a specific task graph. Thus, the PDL model applies to any task graph which follows the object construction scheme specified in the PDL program. This is the essential difference between performance modeling in PDL versus a procedural hardware description language such as VHDL. More information on the PDL language and system, including the system software, can be obtained through the PDL home page on the WWW at http://www.ece.uc.edu/~ ddel/pdl.html.

References

[1] Ranga Vemuri, Ram Mandayam, Vijay Meduri. "Performance Modeling Using PDL". To appear in IEEE Computer, 1995.

[2] Rajesh Kumar Gupta. *"Co-Synthesis of Hardware and Software for Embedded Digitl Systems"*. Kluwer Academic Press, 1995.

[3] Stanford University. *"Protozone: User's Guide"*.

[4] Ramanand Mandayam, Jeffrey Walrath, Ranga Vemuri. *"Performance Description Language Reference Manual"*. University of Cincinnati, 1995.

[5] William B. Pennebaker and Joan L. Mitchell. *"JPEG:Still Image data Compression Standard"*. Van Nostrand Reinhold, 1993.

[6] Gregory K. Wallace. "The JPEG Still Picture Compression Standard". *Communications of the ACM*, pages 30–44, April 1991.

[7] Jay Roy, Nand Kumar, Rajiv Dutta and Ranga Vemuri. "DSS:A Distributed High-Level Synthesis System". In *IEEE Design and Test of Computers*, June 1992.

APPENDIX I:
Performance Verification Using Partial Evaluation and Interval Analysis

Jeffrey Walrath, **Ranga Vemuri**, and **William Bradley**
University of Cincinnati
P.O. Box 210030
ECECS Department, ML. 30
Cincinnati, Ohio 45221-0030

Address for Correspondence:

Dr. Ranga Vemuri, Director
Laboratory for Digital Design Environments
P.O. Box 210030
Department of Electrical and Computer Engineering
University of Cincinnati ML. 30
Cincinnati, Ohio 45221-0030

Phone: (513)-556-4784
Fax: (513)-556-7326
Email: ranga.vemuri@uc.edu

Submitted to: **ED&TC'97** (Accepted)
Category: *9: Formal Verification*

All appropriate clearances for the publication of this paper have been obtained, and if accepted the author will prepare the final manuscript in time for inclusion in the Conference Proceedings and will present the paper at the Conference.

Author _____ Jeffrey Walrath

This work was partially supported by the ARPA RASSP program and monitored by the Wright Lab, US-AF under contract number F33615-93-C-1316 and by the Semiconductor Research Corporation under contract number DJ-293.

Performance Verification Using Partial Evaluation and Interval Analysis

Abstract

Performance models, usually written in high-level programming languages or high-level hardware description languages, make full use of high level procedural constructs such as the assignment statement, if-then, case, while control constructs and procedure calls. We propose a partial evaluation procedure to reduce procedural performance models into an equational form. We then propose an interval-analysis based method to formally determine whether the reduced performance model satisfies a set of relational constraints on the performance attributes. Together, the partial evaluation and interval analysis procedures constitute a powerful approach for formal performance verification. We illustrate this through examples, and describe both techniques in detail. Also included are results for an implementation of a symbolic partial evaluator of performance models and a performance verification tool based on the interval analysis technique.

1 Introduction

System designs consist of a hierarchical collection of modules with ports connected by nets. Performance of a system is described by a collection of attributes attached to the various objects (modules, ports and nets) in the design. A *performance model* is an executable specification where some of these attributes are specified in terms of the other (computed) attributes. Usually, performance models are written in a hardware description language or a high-level programming language using the full power of the procedural programming constructs, such as the assignment statement, conditional and iterative statements and function calls, provided in these languages. In this paper, we refer to these models as *procedural performance models.*

For example, Figure 1 shows a combinational logic design and Figure 2 shows a procedural performance model for computing CMOS dynamic power dissipation based on input signal probabilities [1]. It shows the *primitive* and the *computed* performance attributes. Attributes are attached to each entity in the design and are referenced using the notation *ObjectName'AttributeName.* In this example, all attributes are assumed to be *real* valued. This example is a procedural model due to the presence of function calls which in turn contain variable assignment statements inside while loops.

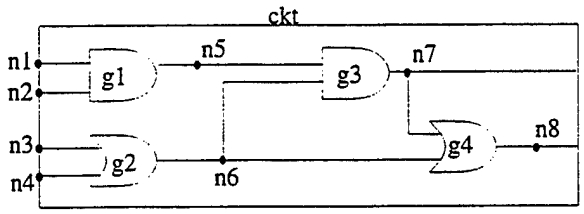


Figure 1: **Example Design Net List**

The *performance verification* problem is to determine whether a performance model can simultaneously satisfy a set of relational constraints placed on the performance attributes. It is known that the performance verification problem is undecidable for procedural performance models [2, 3]. In this paper, we show how *eqauational performance models* can be verified using an interval based analysis technique. An equational performance model consists of equations, one for each computed attribute, in terms of other attributes using a predefined set of mathematical operators. An equational model does not contain any programming constructs such as function calls, conditional and iterative statements, and so forth. Furthermore, as will be discussed in Section 3, mathematical operators in the equations must be *invertible* in the sense that for each operation an inverse operation must exist.

Although operators used in basic expressions in procedural performance models are usually invertible, there are many constructs that are not invertible. For example, variable assignment is non-invertible. Control constructs such as case and while statements and function calls are also non-invertible in the presence of the assignment statement. In some special cases a procedural performance model fragment may be invertible, but that it is invertible is quite hard to determine, requiring detailed mathematical analysis.

The question is how to transform a procedural performance model to an equational model when sufficient primitive attribute data is available. This paper also addresses this question and develops a *partial evaluation* [4, 5] technique to reduce procedural performance models to the equational form. Once reduced, these models can be subjected to formal performance verification using the interval analysis technique. Figure 3 shows the process of performance

Primitive Attributes

ckt'system_freq	ckt'voltage
g1'capacitance	n1'prob
g2'capacitance	n2'prob
g3'capacitance	n3'prob
g4'capacitance	n4'prob

Computed Attributes

```
n5'prob = calc_and_prob[(n1'prob, n2'prob])
n6'prob = calc_or_prob([n3'prob, n4'prob])
n7'prob = calc_and_prob([n5'prob, n6'prob])
n8'prob = calc_or_prob([n6'prob, n7'prob])
g1'freq = min([n5'prob, 1.0 - n5'prob]) * ckt'system_freq * 2.0
g1'power = (ckt'voltage**2 * g1'capacitance * g1'freq) / 2.0
g2'freq = min([n6'prob, 1.0 - n6'prob]) * ckt'system_freq * 2.0
g2'power = (ckt'voltage**2 * g2'capacitance * g2'freq) / 2.0
g3'freq = min([n7'prob, 1.0 - n7'prob]) * ckt'system_freq * 2.0
g3'power = (ckt'voltage**2 * g3'capacitance * g3'freq) / 2.0
g4'freq = min([n8'prob, 1.0 - n8'prob]) * ckt'system_freq * 2.0
g4'power = (ckt'voltage**2 * g4'capacitance * g4'freq) / 2.0
ckt'power = g1'power + g2'power + g3'power + g4'power
```

```
function min(vals[ ])
  begin
    temp := vals[1]
    foreach v in vals
      { if (temp > v) then
        temp := v }
    return temp
  end
```

```
function calc_and_prob(vals[ ])
  begin
    temp := 0
    foreach prob in vals
      { temp := temp * prob }
    return temp
  end
```

```
function calc_or_prob(vals [ ])
  begin
    temp := 0
    foreach prob in vals
      { temp := temp * (1.0 - prob) }
    return 1.0 - temp
  end
```

Figure 2: **Procedural Performance Model for Dynamic Power**

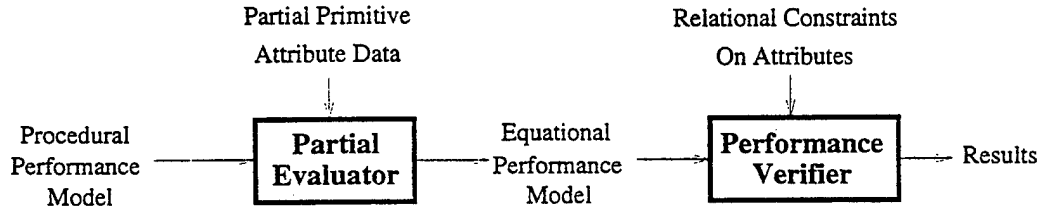


Figure 3: **Performance Evaluation and Verification**

Computed Attributes

```
n5'prob = 0.25
n6'prob = 0.75
n7'prob = 0.1875
n8'prob = 0.796875
g1'freq = 0.5 * ckt'system_freq
g1'power = (ckt'voltage**2 * g1'capacitance * g1'freq) / 2.0
g2'freq = 0.5 * ckt'system_freq
g2'power = (ckt'voltage**2 * g2'capacitance * g2'freq) / 2.0
g3'freq = 0.375 * ckt'system_freq
g3'power = (ckt'voltage**2 * g3'capacitance * g3'freq) / 2.0
g4'freq = 0.40625 * ckt'system_freq
g4'power = (ckt'voltage**2 * g4'capacitance * g4'freq) / 2.0
ckt'power = g1'power + g2'power + g3'power + g4'power
```

Figure 4: **Equational Performance Model for Dynamic Power**

verification using partial evaluation followed by interval analysis.

For example, Figure 4 shows an equational model which is obtained by reducing the procedural model shown in Figure 2 after setting the all input signal probabilities to 0.5 (high and low signal values are equally likely) and partially evaluating the model. This equational model can now be subjected to formal performance verification based on the interval analysis technique.

For example, the question as to whether $10.0 \leq g1'capacitance \leq 25.0$, $5.0 \leq g2'capacitance \leq 10.0$, $5.0 \leq g3'capacitance \leq 20.0$, $8.0 \leq g4'capacitance \leq 15.0$, $1.0 \leq ckt'voltage \leq 5.0$, $10.0 \leq ckt'system_freq \leq 30.0$ implies $0.0 \leq mc'power \leq 12000$ can be answered affirmatively, and the question as to whether $10.0 \leq g1'capacitance \leq 25.0$, $5.0 \leq g2'capacitance \leq 10.0$, $5.0 \leq g3'capacitance \leq 20.0$, $8.0 \leq g4'capacitance \leq 15.0$, $3.3 \leq ckt'voltage \leq 3.5$, $30.0 \leq ckt'system_freq \leq 50.0$ implies $0.0 \leq mc'power \leq 2000$ can be answered negatively once the model is reduced to the equational form. Of course, the verification is valid only within the partial data with which the model was partially evaluated. This approach is analogous to the use of symbolic simulation followed by boolean tautology checking for verifying logic circuits [6].

The rest of this paper is organized as follows: Section 2 introduces a notation for writing procedural performance models and also describes a procedure for the partial evaluation of procedural performance models given partial primitive attribute data. Performance models written using this notation can be easily embedded into high level programming or hardware description languages. Additionally, when sufficient primitive data is available, the reduced models can be rendered in the equational form. Section 3 describes our performance verification technique, based on interval mathematics, for equational models. Section 4 presents experimental results that show typical partial evaluation and verification times for some performance models. Section 5 contains concluding remarks.

2 Partial Evaluation of Procedural Performance Models

Conceptually, a performance model is specified by augmenting a (possibly hierarchical) net-list with attributes and attribute evaluation rules [7, 8]. An attribute represents some design parameter such as voltage, power consumption, time delay, and so forth. An attribute can be either *primitive* or *computed.* Primitive attributes are assigned a value by the user, whereas computed attributes are defined by an *evaluation rule* which assigns an *expression* to the attribute. Evaluation rules can use many different forms of expressions which will be described in the following paragraphs. For uniformity of presentation, we will assume that all attributes are real valued, although the partial evaluation and verification techniques presented in this paper are fully capable of handling integers and enumerated types including booleans and bits.

Figure 5 shows an algorithm for partially evaluating a performance model. $\mathcal{A}_{set}$ is a set containing all of the attributes in the model. Prior to partial evaluation, the *evaluation order* of all the attributes has to be determined. A computed attribute has an expression which defines how to calculate the value of the attribute. This expression typically depends upon the value of other attributes in the performance model. For example, if two rules were $x = y + 5$ and $y = 5$, the rule for x could not be evaluated until y has been. An attribute which depends upon no other attribute is given an evaluation order of 1. From there, each attribute expression is assigned an evaluation order equal to one plus the largest evaluation order of any attribute upon which it depends. In the previous example, y would have an order of one and x would have an order of two.

```
EVALUATE_MODEL(A_set)
begin
   Determine_Evaluation_Order(A_set)
   T_step ← 1
   while(T_step is less than or equal to the largest evaluation order)
      O_set ← {All attributes in A_set with order equal to T_step}
      for each A in O_set
         E ← Evaluation_Expression(A)
         E ← PartialEval(E)
      end for
      T_step ← T_step + 1
   end while
end
```

Figure 5: **Partial Evaluation Algorithm**

Using the evaluation order, each attribute in the model is evaluated beginning with all attributes of order 1. The function *PartialEval*() then performs the process where all *known* attributes values are replaced in each evaluation rule, and evaluation rules are reduced as much as possible. An attribute is considered *known* if it has a single real value. An attribute with an evaluation rule is considered *unknown* until the evaluation rule can be evaluated to a single value.

The following sections describe in detail how to partially evaluate the various constructs. The constructs discussed in this section are available in virtually all high level procedural programming languages and hardware description languages. Performance models can be directly written using such languages, or alternatively, such performance models can be automatically extracted, given the design net-list, from generic performance models written in a performance modeling language such as PDL [7]. Instead of selecting an existing language, we use this general notation to emphasize that the partial evaluation technique described in this paper can be used in the context of performance models written in many existing languages.

Mathematical Expressions : Every mathematical expression is parsed into an expression tree with nodes in the tree representing operations, real values, or references to other attributes. The evaluation process recursively traverses the tree replacing nodes with real values whenever possible.

attr = *unary-operator* (V_1 = *PartialEval(expr)*
attr = (V_1 = *PartialEval(left-expr)*) *binary-operator* (V_2 = *PartialEval(right-expr)*)

If-Then-Else Expressions or statements : As shown below, each part of the if-then-else expression is evaluated first. When the conditional expression is known, the entire if-then-else expression or statement can be replaced by either the true or false branch, depending upon the boolean value of the conditional statement. When the conditional does not evaluate to a known value, the only operations are replacement of all known values where possible.

```
attr = if (V1 = PartialEval(conditional-expr)) then
         (V2 = PartialEval(true-expr))
       else
         (V3 = PartialEval(false-expr))
       endif
```

Case Expressions or statements : This is very similar to the the if-then-else expression or statement. All expressions within the case expression are evaluated. When both the switch expression and matching expression are known, the entire case statement can be replaced with the corresponding arm expression or statement. When this condition does not occur, only values for those attributes which are known can be replaced.

attr = **case** (V_1 = *PartialEval(switch-expr)*) **of**
(V_2 = *PartialEval(match-expr)*) : (V_3 = *PartialEval(arm-expr)*)
(V_4 = *PartialEval(match-expr)*) : (V_5 = *PartialEval(arm-expr)*)
(V_6 = *PartialEval(match-expr)*) : (V_7 = *PartialEval(arm-expr)*)
⋮
others : (V_x = *PartialEval(other-expr)*)
end case

Foreach Expressions or statements : There are two different types of foreach expressions or statements. One type of foreach contains a loop variable that iterates over a range of values from one value to another value by a specified step size. When the left and right range expressions are known, the foreach expression or statement can be unrolled and replaced by copies of the foreach body with the loop variable replaced in each copy with the respective value. When either element of the range is unknown, only references to known attributes in the foreach body can replaced.

attr = **foreach** *var* **in** (V_1 = *PartialEval(left-expr)*) **to**
(V_2 = *PartialEval(right-expr)*) **by** (V_3 = *PartialEval(step-size)*)
{ (V_4 = *PartialEval(body-expr)*) }

The other type of foreach expression iterates over a list of variables, values, or combination of both. Partial evaluation here is similar to the other foreach expression or statement.

attr = **foreach** *var* **in** *iterate-list*
{ (V_1 = *PartialEval(body-expr)*) }

Begin-End Sections : The process for evaluating the begin-end expression begins by setting a temporary fail flag to false. Each variable declaration statement is evaluated along with the initial value if there is one. If any of the variable declaration statements evaluate to unknown, the fail flag is set to true.

Then each programming statement in the begin-end expression is evaluated. If during the evaluation of a statement, the result is unknown, the fail flag is set to true. When a return statement is reached, several conditions are checked. First, the return expression is evaluated. If that value is known and the fail flag is still false, then the entire begin-end expression can be replaced by the residual return expression. However, if the fail flag is true, that means a previous statement did not completely evaluate so the begin-end expression can not be replaced.

Function Calls : Function calls are the most complicated expression to evaluate. First, a copy of the function body (which is a begin-end expression) is made. Then variable declarations are added to the top of the copied function body. For each argument in the function argument list, a declaration is made for that variable and the initial value is set to the value being passed to the function. The following example illustrates this process:

attr = min_val(obj1'val, obj2'val)
function min_val(a, b)

attr = **begin**
a := obj1'val

```
begin                                    b := obj2'val
  ⋮                                        ⋮
end                                      end
```

Once the function call is replaced, the begin-end expression is evaluated. The function call becomes equational only if the residual return expression of the begin-end expression is equational.

By specifying the appropriate partial primitive data for the performance model, all expressions and statements in the performance model can be reduced to an equational form during evaluation. In the case where all primitive data is supplied, the entire performance model becomes evaluated with every attribute having a single real value. This is full evaluation of the model and does not require verification.

3 Verification of Performance Models

Performance verification is the problem of determining whether a performance model can simultaneously satisfy a set of relational constraints on the attributes. Interval mathematics [9, 10, 11] provides a convenient technique to represent relational constraints as intervals. The constraints are specified, the interval technique is applied, and a verification result is produced. This result is in the form of a statement that the constraints can be met ("yes"), or they cannot be met ("no").

However, our approach is limited to performance models that contain only equations. That is, every evaluation rule is only composed of the mathematical operators such as +, -, *, /, x^y, negation, exp(), and log().

Interval Notation: An interval is a tuple of the form $[a, b]$ where $a \leq b$. It denotes the set of all values from a to b, both inclusive. A relational constraint on an attribute is represented by an interval. Figure 6 shows the interval notation for each type of relation that is possible on attribute X. A set of constraints can be imposed on a single attribute with the union of corresponding intervals. For example, the constraint $X < 4$ or $X \geq 6$ would be written as [-∞,4) ∪ [6,∞].

With given a performance model, relational constraints can be placed on various attributes in the performance model. Relational constraints on primitive attributes state the assumptions about the permitted variance in the operating condition of the performance model and the relational constraints on the computed attributes state the desired performance goals.

Initially, each attribute is assigned an initial interval. A computed attribute with an equation or a primitive attribute with no user-specified relational constraints has initial interval of [-∞,∞].

[c,c]	X = c
[-∞,c]	X ≤ c
[c,∞]	X ≥ c
[a,b]	a ≤ X ≤ b
(a,b]	a < X ≤ b
[a,b)	a ≤ X < b
(a,b)	a < X < b
[-∞,c) ∪ (c,∞]	X ≠ c

Figure 6: **Equivalent Relation and Interval**

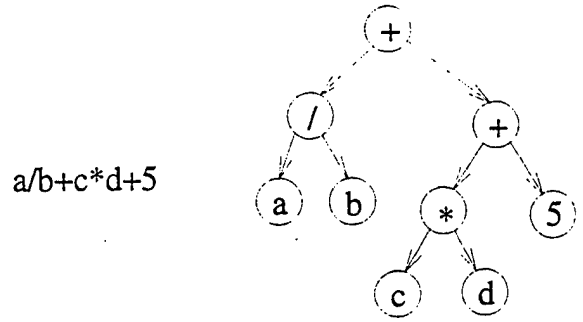


Figure 7: **Example Expression Parse Tree**

Attributes that have a constant real value *val* are assigned an initial interval of [*val*,*val*]. Any constant value appearing in an equation also has an initial interval of [*val*,*val*]. A user specified constraint placed on an attribute replaces the initial interval for the attribute.

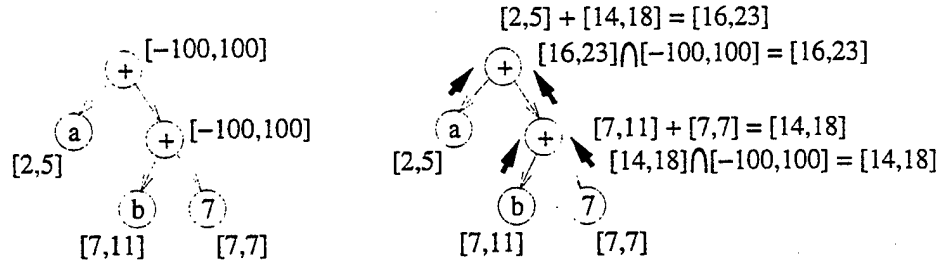
Algorithm for Interval Analysis: To make the explanation of the algorithm clearer, we assume that the attributes have only a single, real-valued interval constraint. A companion paper [12] describes how a variation of this technique can be used to incorporate multiple intervals (multiple relational constraints) for each attribute and integer intervals (including handling of enumerated range intervals).

Before the analysis begins, each equation is parsed into an expression tree (parse tree). Internal nodes in the tree are mathematical operators with edges pointed to to either one or two child nodes depending on whether the operator is unary or binary. The leaves of the tree are either attribute names or constant values. Figure 7 is an expression tree for the equation $x = a/b + c*d+5$. An expression parse tree for the entire performance model is generated in this fashion. The entire performance model is represented as a forest of expression trees.

The interval analysis algorithm makes repeated use of two basic steps, a *forward* interval analysis step followed by a *backward* interval analysis step. In the forward direction, beginning with rules having an evaluation order of 1, each equation is evaluated using interval mathematics. Interval mathematics define how each operator behaves when calculating with intervals. Figure 8 shows each mathematical operator and how to determine a resulting interval.

addition	: [a,b] + [c,d] = [a+c, b+d]
subtraction	: [a,b] - [c,d] = [a-d, b-c]
multiplication	: [a,b] - [c,d] = [min(a*c, b*c, a*d, b*d), max(a*c, b*c, a*d, b*d)]
division	: [a,b] / [c,d] = {[a,b] / [c, 0)} ∪ {[a,b] / (0, d]} *when [c,d] contains*
division	: [a,b] / [c,d] = [a,b] * [1/d, 1/c] *when [c,d] does not contain zero*
minus	: - [a,b] = [-b, -a]
exp()	: exp([a,b]) = [exp(a), exp(b)]
log()	: log([a,b]) = [log(a), log(b)] *when* $a > 0$
log()	: log([a,b]) = UNDEFINED *when* $b < 0$
X^Y	: $[a,b]^{[c,d]}$ = exp([c,d] * log([a,b])) *when* $X \geq 0$
union	: [a,b] ∪ [c,d] = [min(a,c), max(b,d)]
intersection	: [a,b] ∩ [c,d] = [max(a,c), min(b,d)]

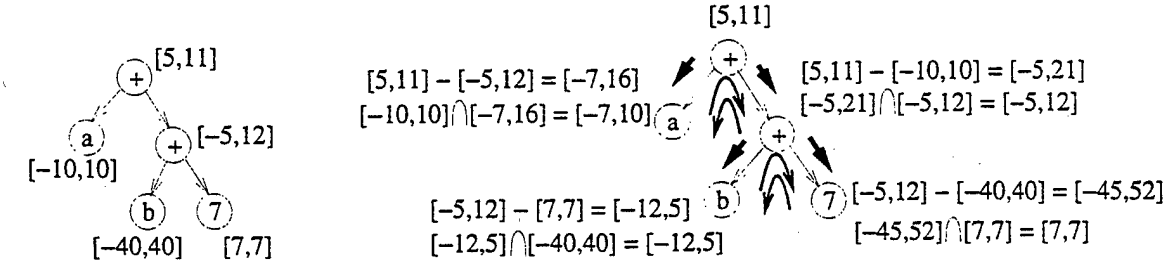
Figure 8: **Mathematical Operators on Intervals**



Before Forward Analysis

After Forward Analysis

Figure 9: **Forward Interval Analysis Example**



Before Backward Analysis

After Backward Analysis

Figure 10: **Backward Interval Analysis Example**

Forward interval analysis of an equation begins by traversing the expression tree from the leaves to the root. The intervals at the leaf nodes are passed to their parent nodes. In the parent node, the appropriate operator is performed and a new interval is created. This new interval is intersected with the current interval at that node to produce the final result. This process is repeated until the interval at the root of the tree is revised. Figure 9 is a simple example that illustrates forward interval analysis on an expression parse tree. Forward (upward) propagation of intervals constitutes computing the parent intervals from the child intervals.

Each equation with an evaluation order of 1 is evaluated in this manner. Next, the equations with evaluation order 2 are analyzed, and this process continues until all equations have been forward analyzed. If at any time an empty or an illegal interval is generated, all analysis stops. An illegal interval is an interval [a,b] where b < a. This "interval" has no values in it and is considered *empty*. Once an interval becomes empty, no further propagation can occur because intersection with an empty interval always produce an empty interval.

The occurrence of an empty interval means that with the given performance model can not simultaneously satisfy all constraints. Thus, analysis stops and the result of verification is that the constraints *cannot* be met; that is the system of constraints can not be satisfied by the model. There is no possible assignment of values to the primitive attributes within the specified ranges that would meet the overall performance goals as stated.

However, after all equations have been forwarded analyzed and no empty intervals were generated, the next step is to do backward interval analysis. In backward analysis, the expression

$X = A + B$	: $A = X - B$ and $B = X - A$
$X = A - B$	: $A = X + B$ and $B = A - X$
$X = A * B$	: $A = X/B$ and $B = X/A$
$X = A\ B$	: $A = X * B$ and $B = A/X$
$X = -A$	: $A = -X$
$X = log(A)$	: $A = exp(X)$
$X = exp(A)$	; $A = log(X)$
$X = A^B$	: $A = exp(log(X)/B)$ and $B = log(X)/log(A)$

Figure 11: **Inverse Calculations**

parse trees are used again. However, evaluation starts at the root and propagates intervals down the tree instead of up the tree as in forward analysis. For each node, a new interval value is calculated using the current interval values of the parent node and the sibling node. This new interval that is calculated is intersected with the current interval value at that node to obtain a new interval value for that node.

To calculate a new interval for a node, the *inverse* of the operator at its parent node must be considered. For example, suppose there is an addition node with an interval X and two children with intervals A and B. In the forward propagation direction the expression would be $X = A + B$. However, in backward propagation, a new interval is calculated for A using $A = X - B$ and a new interval for B is calculated as $B = X - A$. Each node has the computed interval intersected with its current interval, and the algorithm traverses the expression tree until leaf nodes are reached. Figure 10 shows an example of backward analysis for the same expression tree in Figure 9.

Every mathematical operator in the expression trees must have an inverse operator for backward analysis to work correctly. (This, in fact, necessitates the restriction that this technique is applicable to invertible equational performance models only.) Figure 11 shows the inverses for each operator where X is the interval of the current node, A is the interval of the left child and B is the interval for the right child.

Backward analysis continues as long as an empty interval is not produced and until all equations have been backward analyzed. When an empty interval is produced, all analysis stops and the result is that the performance model is unsatisfiable with the given set of constraints. Otherwise, forward and backward propagation are repeated until no further interval changes occur. If this happens, the constraints are satisfiable (ie. there exists a set of values which when applied to the model will produce a solution in the desired range).

Figure 12 is the algorithm for the entire verification process with forward and backward analysis. $\mathcal{N}_{set}$ is the set of all nodes in the expression trees for all expressions in the performance model. Note that the algorithm will always produce a result of either satisfied or unsatisfied. In the case that that an empty interval is generated during iteration, the algorithm ceases and returns a status of unsatisfied. When this does not happen, the outer while loop continues to iterate until no node interval changes during a forward and backward iteration. In theory, it is possible that this may never happen. However, due to the computer's finite precision, there will always be an iteration where no change occurs. In practice, this limit on the precision is small enough that it does not affect the results in a practical performance modeling situation.

```
VERIFY_MODEL(N_set)
begin
   Determine_Evaluation_Order(N_set)
   Done ← false
   while(Done is false)
      Done ← true
      T_step ← 1 /* Forward Propogation */
      while(T_step is less than or equal to the largest evaluation order)
         O_set ← {All nodes in N_set with order equal to T_step}
         for each N in O_set
            I ← Get_Interval(N)
            I_temp ← Perform_Interval_Operation(N)
            I_new ← I ∩ I_temp
            if(I_new is empty) then
               return Unsatisfied
            end if
            if(I not equal to I_new) then
               Replace_Interval(N, I_new)
               Done ← false
            end if
         end for
         T_step ← T_step + 1
      end while
      T_step ← largest evaluation order /* Backward Propogation */
      while(T_step > 0)
         O_set ← {All attributes in N_set with order equal to T_step}
         for each N in O_set
            I ← Get_Interval(N)
            I_L ← Get_Left_Child_Interval(N)
            I_R ← Get_Right_Child_Interval(N)
            I_tempR ← Perform_Inverse_Interval_Operation(I, I_L)
            I_tempL ← Perform_Inverse_Interval_Operation(I, I_R)
            I_newR ← I ∩ I_tempR
            I_newL ← I ∩ I_tempL
            if(I_newL or I_newR is empty) then
               return Unsatisfied
            end if
            if(I_L not equal to I_newL) then
               Replace_Left_Interval(N, I_newL)
               Done ← false
            end if
            if(I_R not equal to I_newR) then
               Replace_Right_Interval(N, I_newR)
               Done ← false
            end if
         end for
         T_step ← T_step − 1
      end while
   end while
   return Statisfied
end
```

Figure 12: **Verification Algorithm**

Constraints

g1'capacitance : [10.0, 25.0]	g4'capacitance : [8.0, 15.0]	ckt'system_freq : [10.0,30.0]
g2'capacitance : [5.0, 10.0]	ckt'power : [0.0,12000.0]	
g3'capacitance : [5.0, 20.0]	ckt'voltage : [1.0,5.0]	

Results **Constraints were satisfiable**

ckt'power : [63.125,11660.2]	g1'power : [25,4687.5]	g3'power : [9.375,2812.5]
	g2'power : [12.5,1875]	g4'power : [16.25,2285.16]

Figure 13: **First Verification Configuration**

Constraints

g1'capacitance : [10.0, 25.0]	g4'capacitance : [8.0, 15.0]	ckt'system_freq : [30.0,50.0]
g2'capacitance : [5.0, 10.0]	ckt'power : [0.0,2000.0]	
g3'capacitance : [5.0, 20.0]	ckt'voltage : [3.3,3.5]	

Results **Constraints were not satisfiable**

ckt'power : [ ]	g1'power : [816.75,3828.13]	g3'power : [306.281,2296.88]
	g2'power : [408.375,1531.25]	g4'power : [530.888,1866.21]

Figure 14: **Second Verification Configuration**

4 Implementation and Results

The partial evaluator and the interval-analysis based performance verification tool are implemented in C++ on Sun Sparc platforms. In the first subsection below, we show the interval constraints and results produced by verification of the reduced equational performance model shown in Figure 4. Three different verification exercises for this model are presented to describe how the verifier can be used. The second section shows evaluation and verification times for two different performance models for large design net-lists.

4.1 Verification of the Performance Model for Power

A constraint configuration or simply *configuration* specifies the relational constraints to be placed on the attributes of a performance model. Figure 13 is one configuration for the primitive attributes in the performance model. Additionally, we constrain *ckt'power* to answer the question: with the given primitive attribute constraints, can the power constraint be satisfied?

The equational model and configuration are given as input to the verifier and two results are produced. First, the verifier specifies whether or not all the constraints were satisfied. In addition, it also lists all the attributes and their last calculated interval values when analysis finished. For the configuration in Figure 13, the constraints were satisfiable. Only the intervals that were different from the original configuration are shown here. Notice that the interval for *ckt'power* has changed from the interval originally specified.

Figure 14 is another configuration with slightly different constraints. In this case, the verifier shows that the the constraints were not satisfiable. Again, only those intervals which are different from the original specification are shown.

A final configuration for the performance model uses a union of intervals for several attributes. Figure 15 shows the intervals separated by commas. A list of intervals separated by commas is equivalent to the union of the those intervals. This configuration was shown to be satisfiable

Constraints

g1'capacitance : [10.0, 25.0]	g4'capacitance : [8.0, 15.0], [3.0,3.5]	ckt'system_freq : [80.0, 90.0],[30.0,50.0]
g2'capacitance : [5.0, 10.0]	ckt'power : [0.0,2000.0]	
g3'capacitance : [5.0, 20.0]	ckt'voltage : [3.3,3.3], [3.0,3.0]	

Results **Constraints were satisfiable**

g1'power : [675,2812.5]	g1'capacitance : [10,25]	ckt'power : [1704.37,2000]
g2'power : [337.5,1125]	g2'capacitance : [5,10]	ckt'voltage : [3,3]
g3'power : [253.125,1687.5]	g3'capacitance : [5,20]	ckt'system_freq : [30,50]
g4'power : [438.75,1371.09]	g4'capacitance : [8,15]	

Figure 15: **Third Verification Configuration**

by the verifier. This time, those attributes which have a union of intervals are shown with the interval that was used during evaluation to produce that satisfiable result.

4.2 Execution Times

We now present results of partial evaluation and verification times for larger performance models. The first performance model was written to calculate the throughput time of combinatorial circuits, given the delay times of each of the gates. A program was written that generated 12 different large combinatorial circuits containing from 1 to 12,286 net-list objects (an object being a single module, port, or net). Using PDL, a performance model for calculating throughput rate was generated for each of the 12 net-lists.

Next, each net-list was partially evaluated, after setting the data arrival time at input ports to '0 ns', to produce an equational model. This model was then verified with a set of constraints that is satisfiable. The same net-list was again verified with a set of constraints that is not satisfiable.

Times for partial evaluation and verification were measured on a Sun SPARCstation 20 containing 256 megabytes of memory. Figure 16 is a plot of all the times for the 12 different net-lists. With this model, it is clear that net-lists with fewer than 1000 objects took an insignificant amount of time to evaluate and verify. However, as the net-list size increased, the verification time increased significantly for the satisfiable constraint set. However, unsatisfiable constraints were verified with a negative in a short amount of time, even for large net-lists.

As a second example, a model for calculating dynamic power in CMOS logic circuits was used for 14 different logic circuits. Net-lists ranged in size from 1 to 49,150 objects. Again, each net-list was partially evaluated to produce an equational model, then verified with a set of satisfiable constraints and a set of unsatisfiable constraints. Figure 17 shows the plot of the times for the various net-lists. In this example, verification of the satisfiable constraints was faster than evaluation and verification with unsatisfiable constraints.

5 Conclusion

This paper presented a partial evaluation technique to simplify procedural performance models and render them in an equational form in which they can be subjected to formal verification using interval analysis. This process is similar to the use of symbolic or trajectory evaluation followed by boolean tautology checking for formal verification of logic circuits [13, 6]. Experimental

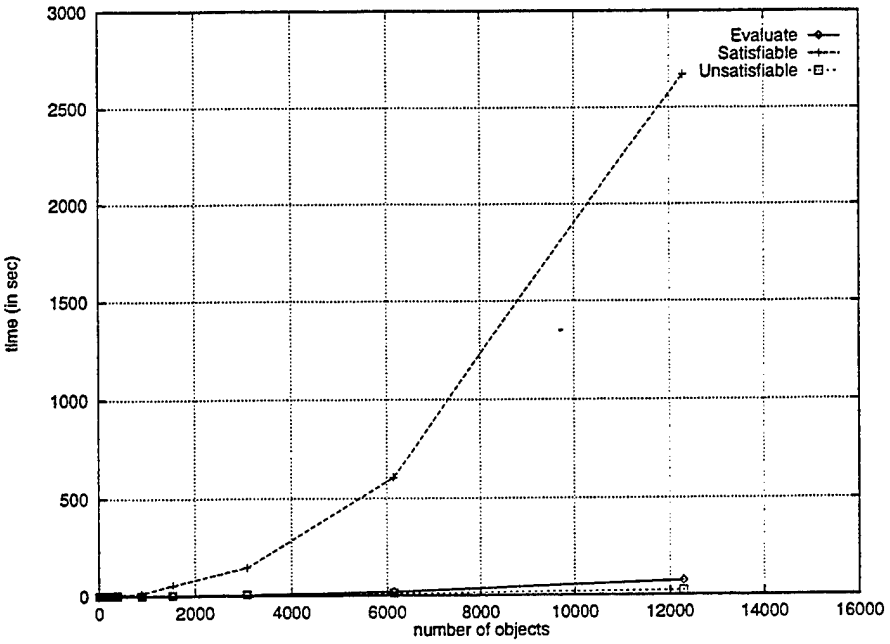


Figure 16: **Evaluation and Verification Times for Delay Model**

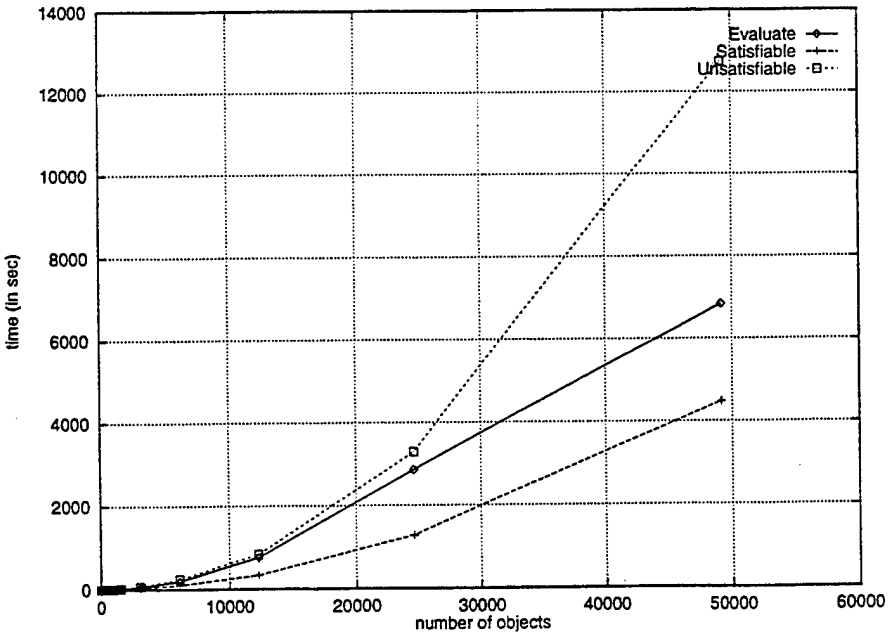


Figure 17: **Evaluation and Verification Times for Power Model**

results show that both the partial evaluation and interval analysis based verification techniques are quite fast even for net-lists contain several thousands of design objects.

We are currently investigating techniques for more closely integrating partial evaluation and interval propagation and for partially evaluating and verifying models that contain dynamic performance attributes that assume streams of values.

References

[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *"Digital Systems Testing and Testable Design"*. Computer Science Press, 1990.

[2] Michael R. Garey and David S. Johnson. *"Computers and Intractability: Guide to the Theory of NP-Completeness"*. W.H. Freeman, 1979.

[3] William Bradley. "Performance Verification of VLSI Systems". PhD Dissertation Proposal, 1995.

[4] Uwe Meyer. Correctness of online partial evaluation for pascal-like language. Technical Report 9205, Justus-Liebig University, October 1992.

[5] Carsten K. Gomard Neil D. Jones and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, Englewood Cliffs, N.J., 1993.

[6] R.E. Bryant. "Can a Simulator Verify a Circuit?". In G. Milne and P.A. Subrahmayam, editors, *Formal Aspects of VLSI Design*, pages 125–136, 1986.

[7] Ranga Vemuri, Ram Mandayam, Vijay Meduri. "Performance Modeling Using PDL". *IEEE Computer*, pages 44–53, April 1996.

[8] Joel M. Schoen, editor. *Performance and Fault Modeling with VHDl.* Prentice Hall, Englewood Cliffs, N.J., 1992.

[9] R.E. Moore. *"Interval Analysis"*. Prentice Hall, Inc., 1966.

[10] William Older and André Vellino. Constraint arithmetic on real intervals. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research.* MIT Press, Cambridge, MA, 1993.

[11] Götz Alefeld and Jürgen Herzberger. *Introduction to Interval Computations.* Academic Press, New York, NY, 1983.

[12] William Bradley and Ranga Vemuri. "Performance Verification Using PDL and Constraint Satisfaction". In *Proceedings of ASP-DAC'95, CHDL'95, VLSI'95*, pages 531–538. CHDL95, 1995.

[13] Scott Hazelhurst and C.H. Seger. "A Simple Thereom Prover Based on Symbolic Trajectory Evaluation and OBDD's". Technical Report TR 93-41, University of British Columbia, 1993.

APPENDIX J:

Hierarchical Behavioral Partitioning for Multicomponent Synthesis

Affiliation: EURO-DAC

Categories: 2.1, 4.5 and 2.8

Abstract

Packaging technology has tremendously improved over the last decade. Various packaging options such as ASICs, MCMs, boards, etc. should be well explored at early stages of the system-synthesis cycle. In this paper we present a hierarchical behavioral partitioning algorithm which partitions the input behavioral specification into a hierarchical structure and binds all elements of the structure to appropriate packages from a given package library. As an application to our partitioner, we integrated the partitioner with a high level synthesis tool to create an environment for multicomponent synthesis and hierarchical package design. We provide detailed partitioning algorithms and experimental results.

1 Introduction

High level synthesis converts a behavioral specification of a digital system into an equivalent RTL design (composed of a data path and a finite state controller; the data path is a composition of components selected from a register-level component library) that meets a set of stated performance constraints [1, 2, 3]. This RTL design can be partitioned into multiple segments to realize a multichip design. Partitioning RTL designs, however, has various drawbacks: (1) Control lines could be crossing segment boundaries; (2) Operators could be shared by operands in different segments, this results in poor performance due to interchip communication; (3) The design is fixed during synthesis and thus there is very little scope for circuit transformations to improve performance; (4) RTL designs are much larger than their behavioral counterparts, thus, the solution space increases rapidly with the size of the synthesized behavior, making the partitioning process very time consuming; and (5) Power estimation/measurement for RTL designs is too time consuming and not viable for very large designs.

Recent efforts in system-level synthesis have led to the development of high level synthesis systems that can produce multichip digital systems [4, 5, 6]. These systems, however, do not consider the impact of packaging on high level synthesis and hence designs produced by these systems cannot efficiently use available high performance packaging technology. For very large, performance critical designs, an efficient *hierarchical behavioral partitioner*, which fully explores various packaging options, is required to tackle the drawbacks of RTL partitioning. The inputs to the *Hierarchical behavioral partitioner* are: (1) a behavioral specification to partition; (2) parameterized register level component library characterized for area, delay, and switching activity; (3) package library with area, pins, switching activity, clock speed, and cost information for all packages; and (4) cost constraint C, in dollars on the entire design. The output of the partitioner is: (1) a set of behavioral specifications, which together form the original specification; (2) a set of structures that realizes the hierarchical design; and (3) a binding of the behavioral specifications and the structures to appropriate cost effective packages from the package library.

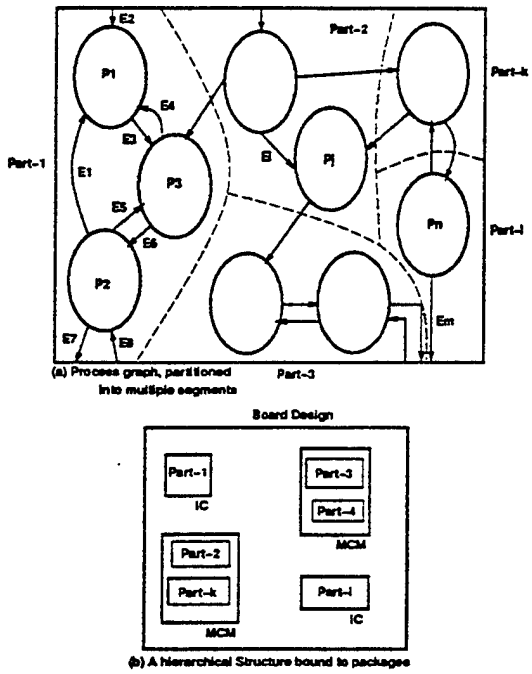


Figure 1: **Hierarchical Behavioral Partitioning**

The input behavioral specification (which may be given in VHDL) consists of a set of communicating and

concurrently executing processes. This specification is internally represented as a process graph; with nodes in this graph representing concurrently executing processes, and edges being communication channels Figure 1 shows a process graph and its hierarchical partition. All multiple-process segments in the figure marked Part-i are behavioral specifications themselves and can be synthesized into register level designs. All these register level designs together with the global controller form the multicomponent design. The hierarchical design mapped onto packages shown in the board design forms a package hierarchy for the design. We formulate the hierarchical partitioning problem and and propose a solution for the hierarchical partitioning and package binding problem. We show how our partitioner can be integrated with a high level synthesis tool to create an environment for multicomponent synthesis and hierarchical package binding. Experimental results for a number of designs are presented.

2 Problem Formulation

Definitions 2.1 and 2.2 introduce the concept of a hierarchical k-level partition of a set. Definition 2.3 extends our notion of a k-level partition of a set to a k-level partition of a graph $G = (N, E)$ (which in our case is a *process graph*), where N is the set of nodes and E is the set of edges.

Definition 2.1 A *1-level partition* of a set $\mathcal{N}$ is a collection, $\mathcal{S}$, of nonempty sets (called segments), such that
- $\mathcal{S}$ is a collection of mutually disjoint sets, i.e., if $C \in \mathcal{S}, D \in \mathcal{S}$, and $C \neq D$, then $C \cap D = \phi$, and
- the union of $\mathcal{S}$ is the whole set $\mathcal{N}$, i.e., $\bigcup_{s \in \mathcal{S}} s = \mathcal{N}$. □

Definition 2.2 A *k-level partition*, $\mathcal{P}$, of a set $\mathcal{N}$ is a set of 1-level partitions $P_1, P_2, \ldots, P_k$ such that
- for $1 \leq i < k$, P_{i+1} is a 1-level partition of P_i, and
- P_1 is a 1-level partition of $\mathcal{N}$. □

Definition 2.3 A k-level partition of a graph $G = (N, E)$ is a k-level partition of N, where N is the set of nodes and E is the set of edges. □

The performance attributes of the nodes in the graph G and level 1 partition segments (each segment is viewed as a *sub-graph* of G or a subset of processes in the behavioral specification) in the graph are determined through scheduling and performance estimation of individual nodes or segments [12, 13, 15]. Thus for any *segment*, $s \in P_1$, the performance attributes $A(s)$, $H(s)$, $T(s)$, and $B(s)$ (area, switching activity, clock period and pin count respectively) are computed by the performance estimator built into the partitioning environment. This process is similar to the scheduling and performance estimation steps in high level synthesis [12, 15].

We have a set of packages $p_1, p_2, p_3 \ldots p_n$ in a package library $\mathcal{L}$. Each package p has six attributes: $A(p)$, the area capacity; $H(p)$, the maximum switching activity; $T(p)$, period of the fastest clock allowed by the package; $B(p)$, the number of pins available in p; $C(p)$, the dollar cost of p; and $L(p) \geq 1$ is the level number of the package p. Level of a package is the level in the packaging hierarchy at which the package can be used. All bare-die packages are level one, ASICs and MCMs are level two, boards are level three, and so on. The *defining level* of a library is the smallest k such that no package in the library has level greater than k. For $i > 1$, packages with level i can contain only packages with level $i-1$ and level 1 packages contain the nodes and segments of the process graph. The hierarchical partitioner assigns a package $p \in \mathcal{L}$ to each partition segment in $P_1, P_2, \ldots P_k \in \mathcal{P}$. All packages can be instanced multiple times, that is, two different segments can be assigned the same package type. All segments in P_i, called the *level i segments*, can be assigned only to a package of level i. If p and q are two package instances then, $p \prec q$ denotes 'p contains q'.

Definition 2.4 For any instance, p, of a package from the package library $\mathcal{L}$:

If $2 \leq L(p) \leq k$:

(a) area cost of the package $a(p) = \sum_{p \prec q} a(q)$

(b) heat cost of the package $h(p) = \sum_{p \prec q} h(q)$

(c) pin cost of the package $b(p) =$

$\sum_{e \in E} e$, e spans package instances p_a and p_b; such that: $(L(p_a) = L(p_b) = L(p) - 1) \wedge (p \prec p_a) \wedge (p \not\prec p_b)$

(d) clock period cost $t(p) = max_{p \prec q}(t(q))$

When $L(p) = 1$, the scheduler and performance estimator will determine the above costs based on the *level 1 segment* in p. □

Hierarchical Partitioning Problem: Given a process graph, $G = (N, E)$, a package library $\mathcal{L}$ with defining size k, and a cost constraint C:
- find a (k-1)-level partition $\mathcal{P} = \{P_1, P_2, \ldots, P_{k-1}\}$ of G
- *Let* $P_k = \{s_k\}$; *where*, $s_k = \{s_{k-1} \mid s_{k-1} \in P_{k-1}\}$ that is, P_k contains exactly one segment (which in turn contains all the segments in P_{k-1}) to be mapped to a top most level package in the library.
- Now find a binding, $\mathcal{B}$, which for $1 \leq i \leq k$, binds each segment in P_i to some level i package instance from $\mathcal{L}$, such that

for each instance, p, of any package from $\mathcal{L}$:

$a(p) \leq A(p)$,
$h(p) \leq H(p)$,
$b(p) \leq B(p)$,
$t(S) \geq T(p)$.

subject to

$$Cost(\mathcal{P}) = \sum_{instance\ p} C(p);\ Cost(\mathcal{P}) \leq \mathcal{C}. \qquad \square$$

3 The Behavior Level Hierarchical Partitioning Algorithm

The algorithm begins by partitioning the process graph and mapping partition segments onto available bare-die packages. A graph is constructed from the partition generated at this level for further partitioning at the next higher level of packaging. The packaged partition segments form nodes in the new graph; edges of the current graph which connect nodes in different segments, form the edges of the new graph. At the next higher level of packaging, this new graph is partitioned and mapped onto packages. This process continues until the packaging hierarchy is exhausted and at each level, partition segments are mapped onto cost effective packages. If, at a particular level, no solution is found, we back-track to the previous level, tighten cost constraints, and construct a new partition and continue. Various steps in the algorithm are explained below.

Setting Constraints: Initially, on the first pass, overall area and switching activity constraints for the entire design are set to the minimum area and switching activity capacity of packages at the highest level in the package hierarchy (since, eventually, the design hierarchy needs to be mapped onto a package at the topmost level in the package hierarchy). The cost constraint is set by subtracting the cost of the smallest package at all levels of packaging above level 1 from the total cost constraint, $\mathcal{C}$. On subsequent invocations, if the algorithm is back-tracking, a cost overrun is computed. If the cost overrun is less than the cost of the previous level's packaging, cost constraint for the previous level (on a back-track) is set by subtracting the product of cost overrun and a *cost overrun factor* (COF < 1) from the cost of the previous level's packaging. On the other hand, if the cost overrun is greater than the cost of the previous level's packaging, cost constraint for the previous level (on a back-track) is set by multiplying the cost of the previous level's packaging by a *constraint tighten factor* (CTF < 1). COF and CTF dictate the rate at which the cost constraint is tightened on a back-track. Typical values of COF are between 0.2–0.3 and CTF between 0.9–0.95 to enable effective search of the design space. If the algorithm is not back-tracking, cost constraint is generated by subtracting the actual cost of packaging at lower levels of packaging and the projected packaging cost at higher levels (cost of smallest packages) from the total cost constraint, $\mathcal{C}$.

Hierarchical Partitioning and Package Design (HPP): Algorithm 3.1 presents the hierarchical partitioning and package design algorithm (HPP). HPP has access to a multiway partitioning algorithm (MP – Algorithm 3.2). When partitioning at any level, HPP first determines cost, area, and switching activity constraints using Set_Constraint and then MP is invoked. MP explores the design space by constructing a set of alternative partitions; MP returns the first partition that satisfies constraints, or, in the absence of a constraint satisfying solution, returns the best cost solution from the set of partitions.

MP returns a *status* flag along with a solution (partition with segments bound to packages). *Status* takes three values of *SUCC, BEST,* or *FAIL* to describe the cases where a constraint satisfying solution is found (a constraint satisfying partition with partition segments mapped onto packages from the package library), a solution is found (valid partition – a partition with segments mapped onto packages, but does not satisfy constraints), or no solution is found (no valid partition – one or more partition segments cannot be mapped onto packages). Based on the values of the *status* flag for the current and previous levels, HPP decides to proceed to the next higher level, back-track to previous level or terminate reporting failure. A *hierarchical netlist manager* (HN) is used to generate a netlist, of the newly generated partition, for use at the next higher level.

Multiway Partitioning Algorithm (MP): MP (Algorithm 3.2) is built on top of a K-way extension of the Fiduccia-Mattheyses algorithm (KWAY – Algorithm 3.3) [11, 14]. MP first determines the minimum and maximum number of segments that *feasible* partitions can have and invokes the KWAY algorithm to generate partitions in the feasible range. MP returns with status SUCC if a constraint satisfying partition is found. When a constraint satisfying solution is not found, MP returns the best solution found with status BEST. In the case of no valid partitions (one or more partition segments cannot be packaged), MP returns FAIL.

K-way FM Algorithm (KWAY): Our *k-way* extension of the FM algorithm (KWAY — Algorithm 3.3) starts by creating a random initial partition of k segments. k-way partitioning is carried out by repeatedly invoking *two-way* FM (*two_way_fm*) on pairs of partition segments. *two_way_fm* tries to improve bi-partitions by moving one node at a time

Algorithm 3.1 (HPP Algorithm: HierPartPack)
G: input graph (Behavioral specification)
P: package set
C: overall cost constraint on design
HN: hierarchical netlist manager
StatArr[k], BtkArr[k]: status of partitioning and number of back-tracks at each level
MaxBtk: User specified limit on number of back-tracks at any level
k: levels in package hierarchy, level: current level
area: overall area constraint
switch: overall switching activity constraint
cost: cost constraint at current package level

```
HierPartPack(G, P, C)
begin
  level ← 1      G_level ← G      Solution ← null
  while level < k do
     Set_Constraint()
     (status, Solution) ← MP(G_level, P(level), cost,
              area, switch, level)
     StatArr[k] ← status
     case status is
        SUCC:
           level ← level + 1
           HN :: read_partition(Solution)
           HN :: construct_netlist(level)
        BEST:
          if ((StatArr[level − 1] = SUCC) ∧
            (BtkArr[k] < MaxBtk)) then
             BtkArr[k] ← BtkArr[k] + 1
             level ← level − 1     /* back-track */
          else
             level ← level + 1
             HN :: read_partition(Solution)
             HN :: construct_netlist(level)
          end if
        FAIL:
          if ((StatArr[level − 1] = SUCC) ∧
            (BtkArr[k] < MaxBtk)) then
             BtkArr[k] ← BtkArr[k] + 1
             level ← level − 1     /* back-track */
          else
             return(null)
          end if
     end case
     G_level ← HN :: read_netlist(level)
     /* retrieve next level netlist */
  end while
  return(Solution)
end
```

Algorithm 3.2 (Multiway Partitioning Algorithm)
G: input graph, P: package set
p: individual package from P
area: overall area constraint
switch: overall switching activity constraint
C: cost constraint on design
level: level in package hierarchy

```
MP(G, P, C, area, switch, level)
begin
     min_seg ← max(area/max_area(p),
switch/max_switch(p))
  max_seg ← num_cell(G) /* # of nodes in graph */
  best_cost ← ∞      status ← FAIL
  Solution ← null
  for num_seg = min_seg to max_seg do
     Best ← KWAY(G, P, num_seg, level)
     /* generate first partition */
     num_fm_ite ← 1      num_fm_imp ← 1
     status ← check_constraint(Best, area, switch, C)
     while (status ≠ SUCC ∧
       num_fm_ite < MAX_FM_ITE ∧
       num_fm_imp < MAX_FM_IMP)  do
         S ← KWAY(G, P, num_seg, level)
         status ← check_constraint(S)
         num_fm_ite ← num_fm_ite + 1
         best_cost_kway ← cost(Best)
         if (status = SUCC) ∨ ((status = BEST) ∧
           (cost(S) < best_cost_kway)) then
             Best ← S
         end if
         if (cost(S) < best_cost_kway) then
           num_fm_imp ← 1
         else
           num_fm_imp ← num_fm_imp + 1
         end if
     end while
     if status = SUCC then
        return (status, Best)
     elsif (status = BEST) ∧ (cost(Best) <
          best_cost) then
        Solution ← Best
        best_cost ← cost(Best)
     end if
  end for
  return(status, Solution)
end
```

Algorithm 3.3 (k-way FM Algorithm: KWAY)
G: graph $G = (V, E)$, V is a set of vertices and E is a set of edges
P: set of packages, S: $\{s_1, s_2, \cdots, s_n\}$ a partition of G with k segments

```
KWAY(G, P, k, level)
begin
  Best ← initialize()   /* create initial partitions */
  if level = 1 then   /* pure behavior specification
                - estimate attributes */
     for all s ∈ Best do
        Schedule/Performance Estimate s
        and generate A(s), H(s), B(s), and T(s)
     end for
  end if
  best_cost ← 0    S ← null    cont_part ← TRUE
  ite_cnt ← 1    imp_cnt ← 1
  for all s ∈ Best do     /* map partition segment
                to package and find cost */
     best_cost ← best_cost + cost(B(s))
  end for
  while cont_part = TRUE do
     for i = 1 to k−1 do
        for j = i+1 to k do
           two_way_fm(s_i, s_j)
        end for
     end for
     if level = 1 then   /* pure behavior specification
                - estimate attributes */
        for all s ∈ S do
           Schedule/Performance Estimate s
           and generate A(s), H(s), B(s), and T(s)
        end for
     end if
     curr_cost ← 0
     for all s ∈ S do     /* map partition segment
                to package and find cost */
        curr_cost ← curr_cost + cost(B(s))
     end for
     ite_cnt ← ite_cnt + 1
     if curr_cost < best_cost then
        imp_cnt ← 1      Best ← S
        /* save best partition seen so far */
     else   imp_cnt ← imp_cnt + 1   end if
     if ite_cnt = MAX_ITE ∨ imp_cnt = IMP_CNT
       then   cont_part ← FALSE   end if
  end while
  return(Best)          /* retrieve best partition */
end
```

from one partition segment to the other, taking care not to violate area and switching activity constraints. The *two_way_fm* algorithm is based on Fiduccia and Mattheyses's bi-partitioning algorithm [11]. *two_way_fm* is invoked until, either a user specified limit on number of total iterations is exceeded, or a user specified limit on number of iterations over which partition cost does not improve is exceeded. The best cost solution found during the iterations is returned as the k-way partition.

Scheduling and Performance Estimation: To evaluate the cost of level 1 partition segments, the K-way FM invokes the scheduler, which also estimates the performance attributes. Scheduling is the first important step in the high level synthesis process. The input behavioral specification is converted into an equivalent data flow graph (DFG) representation. Scheduling operates on the DFG. DFG operations are assigned to specific control steps and are bound to physical ALUs available in the component library. The output of scheduling is a time-stamped and partially bound data flow graph, that satisfies specified constraints. Scheduling determines execution speed of the synthesized design in terms of clock speed and number of clock cycles required to execute all operations. For a given parameterized component library, we can compute the area, average switching activity, and clock speed costs from the schedule produced by the scheduler. An implementation of Paulin's *force-directed list scheduling* [9], extended for communicating and concurrently executing processes [8], is used. Switching activity estimation technique has been reported in [7].

4 Multicomponent Synthesis

Multicomponent synthesis is carried out by synthesizing individual partition segments at level 1. Figure 2 demonstrates how we integrate our hierarchical partitioning environment with a high level synthesis system to produce multicomponent designs with packaging hierarchy. We call this integrated system, MSS (Multicomponent Synthesis System) [10]. Design tradeoffs are performed by considering various partitions and carrying out scheduling and performance estimation on proposed partition segments. The performance attributes of the synthesized RTL designs are determined and compared against the capacity and cost constraints imposed by the packages they are bound to. Also, a *global controller* is automatically placed on a partition segment and interconnected with the RTL design segments. The global controller is placed on a partition segment whose package has the most space to fit the controller. Details of the controller model to support multicomponent partitioning are discussed in [13, 14, 16].

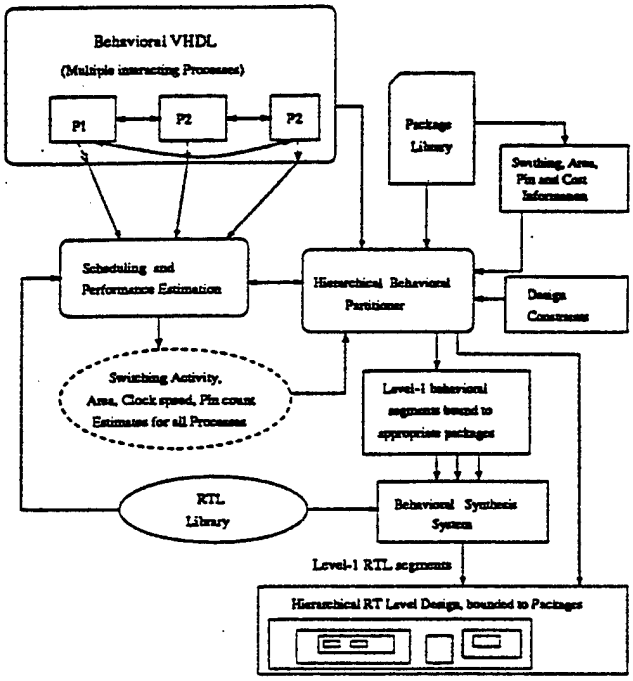


Figure 2: **Hierarchical Behavioral Partitioning for Multicomponent synthesis**

At the end of multicomponent synthesis and hierarchical package design we have a multicomponent design composed of interacting RTL design segments. The behavioral partitioning phase produces multiple behavior segments that are completely synthesized to RTL designs using a high level synthesis system such as DSS [12, 13]. Also produced is a hierarchical structural design (the leaf nodes in this design are the individual RTL designs) that is mapped onto efficient cost-effective packages from a package library. We functionally validate our approach by simulating the hierarchical RTL design and the input behavior for the same set of test vectors and comparing their outputs.

5 Results

We present results for a number of examples to demonstrate the validity of our behavioral partitioning approach for multicomponent synthesis and hierarchical package design. Details of our package library is shown in Table 1. Data about area, pin, switching activity, and clock speed constraints supported by each package and the cost of the package are presented. Table 2 presents details of the number of lines of code in behavior level VHDLspecification and the number of processes for each of our examples.

Move Machine: The instruction set of the Move Machine controls instruction and data flow. It does not compute any data values. ALU operations are assumed to be memory mapped. **Fifo:** Fifo models a producer consumer problem. **Shuffle:** The Shuffle is a high speed reconfigurable 32 bit shuffle-exchange network for parallel signal processing. The Shuffle exchange is a commercial product of Texas Instruments, Inc. dyn is a five process description that monitors and maintains the dynamic length and maximum length to which a queue in a producer-consumer problem grows. alu is a nine process description of an arithmetic logic unit. *dyn1-dyn10* and *alu1-alu5* are multiple processing elements generated by making multiple instantiations of dyn and alu respectively.

L(p)	Name	A(p)*	B(p)	H(p)+	T(p)-	C(p)#
1	Tiny1	5	40	50	50	400
1	Tiny2	5	40	60	50	500
1	Tiny3	8	40	80	50	600
1	Tiny4	12	40	120	50	700
1	Small1	15	40	150	50	800
1	Small2	18	40	200	50	900
1	Small3	20	40	200	50	1000
1	PGA-1	12	84	200	50	1200
1	PGA-2	15	84	300	50	1300
1	PGA-3	18	84	400	50	1400
1	PGA-4	20	84	500	50	1500
1	PGA-5	20	84	800	50	1600
1	PGA-6	20	169	1000	50	1800
2	Pl-1	6	40	50	50	250
2	Pl-2	6	40	60	50	300
2	Pl-3	8	40	80	50	350
2	Pl-4	12	40	120	50	400
2	Pl-5	15	40	150	50	450
2	Cer-1	15	40	200	50	500
2	Cer-2	18	40	250	50	550
2	Cer-3	20	40	300	50	600
2	PGA-1C	12	84	220	50	800
2	PGA-2C	15	84	320	50	900
2	PGA-3C	18	84	450	50	1000
2	PGA-4C	20	84	850	50	1200
2	PGA-5C	20	169	1000	50	1500
2	MCM-1	200	169	1000	75	10000
2	MCM-2	300	169	2000	75	15000
2	MCM-3	400	169	3000	75	20000
3	Board-1	300	80	2000	100	300
3	Board-2	400	80	3000	100	400
3	Board-3	500	128	4000	100	500
3	Board-4	600	128	5000	100	600
3	Board-5	800	128	8000	100	800
3	Board-6	1000	128	12000	100	1200

* : sq. mm; + : 1000 node switches; - : ns; # : $

Table 1: **Package Alternatives**

5.1 Multicomponent Synthesis and Hierarchical Package Design

Tables 3 and 4 present results of multicomponent synthesis and hierarchical package design for the design examples in Table 2 with the package library shown in Table 1. For the smaller examples (Move Mc - dyn2), Table 3 presents: (1) number of processes; (2) hierarchical partition segments mapped onto packages

Example	Num Lines (VHDL)	Num Proc
Mv Mc	75	3
Fifo	65	3
Shuffle	472	5
dyn1	132	5
dyn2	254	10
dyn3	376	15
dyn4	498	20
dyn5	620	25
dyn6	742	30
dyn7	864	35
dyn8	986	40
dyn9	1108	45
dyn10	1230	50
alu1	100	9
alu2	188	18
alu3	276	27
alu4	364	36
alu5	452	45

Table 2: Design Data for Examples

from the package library (at level 1, partitioning of processes into segments, synthesized eventually into RTL designs); (3) actual number of back-tracks by the hierarchical partitioning and package design algorithm and the limit on number of back-tracks (BTK); (4) actual cost of the design and the cost constraint; and (5) execution time. With a large number of processes it is difficult to present assignment of processes to partition segments, hence for dyn3 - dyn4, Table 3 presents the number of processes on each level 1 partition (instead of presenting individual partitions). With an even larger number of processes, it is difficult to present even details of level 2 partition segments. Thus, Table 4 presents the following data for all designs in Table 2: (1) number of processes; (2) number of back-tracks/BTK; (3) actual cost/constraint; and (4) execution time.

For each example, the cost constraint was progressively tightened until the algorithm failed to find a cost-satisfying solution. In all cases, if a constraint-satisfying solution existed, it was discovered by the algorithm. This was verified by manual examination of the examples. The results establish the validity of the algorithm. An interesting observation that vindicates our choice of the back-tracking algorithm is that in all our examples the most times the algorithm ever back-tracks is three (Table 4). This is because the algorithm back-tracks only if it can potentially find a solution with better cost and, also, the algorithm converges to a constraint-satisfying solution fairly rapidly.

Multicomponent Synthesis vs Hierarchical RTL Partitioning: We also developed a Hierarchical RTL partitioner [14] as an alternative approach. Here, we synthesize the input behavior and the partition the resultant RTL design. Table 5 presents a comparison of hierarchical behavioral partitioning and hierarchical RTL partitioning approaches. Blanks indicate that the input design was too large to be handled by the RTL partitioner. For each example, the better dollar cost solution is bold-faced. RTL partitioning yields better designs for smaller examples where the number of synthesized RTL components is relatively small ($<$ 200). For larger examples multicomponent synthesis clearly out-performs RTL partitioning in the quality of solutions. Also, the time taken by RTL partitioning is more than the time taken by multicomponent synthesis by an order of magnitude (two orders of magnitude for larger examples - alu4, dyn3).

Hierarchical Package Design without Scheduling: Since scheduling and performance estimation are time consuming, we modified HCP and KWAY by replacing the schedule and performance estimation steps by approximations for area and switching activity. In this approach, individual processes are first scheduled and performance estimated. Then, for level 1 segments, the area and switching activity costs of the individual processes in the segment are summed to obtain the total area and switching activity of the overall segment. These numbers are then adjusted by a small percentage (10-30%) to take into account the possible sharing of resources if the processes had been actually scheduled together[14]. Table 6 presents results of hierarchical partitioning and package binding with and without an integrated scheduling and performance estimation step.The better dollar cost for each example is bold-faced. *Invalid* indicates that at least one of the partition segments at level 1 does not fit on available packages; thus, the design is not valid. The approach with scheduling out-performs the approximation method, especially for the larger designs. However, (a) execution time for the approximation method is very small; and (b) the estimated cost of packaging the designs are fairly close to the solutionsreported by the algorithm with embedded scheduling algorithm. Thisobservation indicates that the approximation algorithm should be usedto quickly generate approximate dollar cost constraints to be imposed on the rigorous algorithm.

6 Conclusions and Discussion

We have presented a hierarchical behavioral partitioning and package design algorithm. We demonstrated a methodology to integrate our partitioner with a high level synthesis tool to create a multicomponent synthesis and hierarchical package design en-

Example	No. of Procs	Segments and Mapping (s_i–p_i) Level-3	Level-2	Level-1	Num BkTrk/ BTK	Cost/ Constraint ($)	Exec Time (s)
Mv Mc	3	s_{21}–Board-1	s_{11}–PGA-5C	s_1–PGA-6 EXE	1/10	5600/5000	6
			s_{12}–PGA-1C	s_2–PGA-1 FET, DEC			
Fifo	3	s_{21}–Board-1	s_{11}–PI-5	s_1–Small1 FIFO PRODUCER CONSUMER	0/10	1550/3000	2.7
Shuffle	5	s_{21}–Board-2	s_{11}–PGA-4C	s_1–PGA-4 shuffle-1	0/10	13900/12000	59.8
			s_{12}–PGA-4C	s_2–PGA-4 shuffle-2			
			s_{13}–PGA-4C	s_3–PGA-4 shuffle-3			
			s_{14}–PGA-4C	s_4–PGA-4 shuffle-4			
			s_{15}–PGA-4C	s_5–PGA-4 output			
dyn1	5	s_{21}–Board-1	s_{11}–Cer-3	s_1–Small3 s1_p_1,s1_p_pt s1_p_s1,s1_p_2 s1_p_st	1/10	1900/2000	3.6
alu1	9	s_{21}–Board-1	s_{11}–Cer-2	s_1–PGA-1 s1_nbp,s1_nap s1_np,s1_outp	1/10	3100/2500	100.7
				s_2–Tiny1 s1_mp,s1_ap s1_op			
			s_{12}–PI-1	s_3–Tiny1 s1_dp,s1_sp			
dyn2	10	s_{21}–Board-1	s_{11}–Cer-3	s_1–Small-1 s2_p_s1,s2_p_pt s2_p_2	2/10	3350/3200	212.7
				s_2–Tiny1 s2_p_st,s1_p_st			
			s_{12}–PI-5	s_3–Small1 s1_p_s1,s1_p_pt s1_p_1,s1_p_2			
dyn3	15	s_{21}–Board-1	s_{11}–PI-3	s_1–Tiny-3 3 procs	1/10	5000/5000	126.1
			s_{12}–PI-5	s_2–Small1 4 procs			
			s_{13}–PI-5	s_3–Small1 4 procs			
			s_{14}–PI-5	s_4–Small1 4 procs			
alu2	18	s_{21}–Board-1	s_{11}–PGA-3C	s_1–PGA-3 6 procs	1/10	6700/5000	412.8
			s_{12}–PI-5	s_2–Small1 5 procs			
			s_{13}–PGA-2C	s_3–Tiny1 1 proc			
				s_4–Tiny1 3 procs			
				s_5–Tiny1 2 procs			
			s_{14}–PI-1	s_6–Tiny1 1 proc			
dyn4	20	s_{21}–Board-1	s_{11}–PI-5	s_1–Small1 5 procs	0/10	6350/8000	229.3
			s_{12}–PI-1	s_2–Tiny1 1 proc			
			s_{13}–Cer-2	s_3–Small2 6 procs			
			s_{14}–PI-3	s_4–Tiny3 3 procs			
			s_{15}–PI-4	s_5–Tiny4 4 procs			
			s_{16}–PI-1	s_6–Tiny1 1 proc			

Table 3: **Multicomponent Synthesis with Hierarchical Package Design Results**

Note: s–p denotes the mapping of segment s onto package p from the package library. Also, at level 1, number of processes on each partition segment are presented.

Example	No. of Procs	Num BkTrk/ BTK	Cost/Constraint ($)	Exec Time (s)
dyn5	25	0/10	8350/8000	349.5
alu3	27	0/10	12700/8000	579
dyn6	30	1/10	9850/9000	1470.7
dyn7	35	2/10	11200/10000	3141
alu4	36	3/10	14100/15000	1549.4
dyn8	40	1/10	11850/12000	1863.5
dyn9	45	1/10	13800/13000	3684.1
alu5	45	2/10	17750/18000	1626.4
dyn10	50	2/10	16850/15000	6452.2

Table 4: **Multicomponent Synthesis and Package Design Results (Contd ...)**

Example	Num Proc	Num RTL Comp	Hierarchical Behavioral Partitioning			Hierarchical RTL Partitioning			Cost ($) Constr.
			Btk/ BTK	Cost ($)	Exec Time (s)	Btk/ BTK	Cost ($)	Exec Time (s)	
Mv Mc	3	53	1/10	5600	6	0/10	4250	13.2	5000
Fifo	3	76	0/10	1550	2.7	0/10	1750	6.4	3000
Shuffle	5	379	0/10	13900	59.8	-	-	-	12000
dyn1	5	128	1/10	1900	3.6	0/10	1550	11.9	2000
alu1	9	65	1/10	3100	100.7	0/10	1900	6.5	2500
dyn2	10	234	2/10	3350	212.7	0/10	6200	6560	3200
dyn3	15	334	1/10	5000	126.1	0/10	53000	113272	5000
alu2	18	123	1/10	6700	412.8	0/10	5400	2976	5000
dyn4	20	-	0/10	6350	229.3	-	-	-	8000
dyn5	25	-	0/10	8350	349.5	-	-	-	8000
alu3	27	161	0/10	12700	579	0/10	10850	6251	8000
dyn6	30	-	1/10	9850	1470.7	-	-	-	9000
dyn7	35	-	2/10	11200	3141	-	-	-	10000
alu4	36	205	3/10	14100	1549.4	0/10	53600	109850	15000
dyn8	40	-	1/10	11850	1863.5	-	-	-	12000
dyn9	45	-	1/10	13800	3684.1	-	-	-	13000
alu5	45	-	2/10	17750	1626.4	-	-	-	18000
dyn10	50	-	2/10	16850	6452.2	-	-	-	15000

Table 5: **Behavioral Partitioning vs RTL Partitioning approaches**

Example	Num Proc	With Scheduling			Without Scheduling			Cost ($) Constr.
		Btk/ BTK	Cost ($)	Exec Time (s)	Btk/ BTK	Cost ($)	Exec Time (s)	
Fifo	3	0/10	1550	2.7	0/10	1550	1.1	3000
Shuffle	5	0/10	13900	59.8	0/10	13900	29.8	12000
dyn1	5	1/10	1900	3.6	0/10	1900	1.4	2000
alu1	9	1/10	3100	100.7	0/10	3550	11.3	2500
dyn2	10	2/10	3350	212.7	1/10	3600	9	3200
dyn3	15	1/10	5000	126.1	0/10	Invalid	5.8	5000
alu2	18	1/10	6700	412.8	1/10	6800	76.2	5000
dyn4	20	0/10	6350	229.3	0/10	7150	10.3	8000
dyn5	25	0/10	8350	349.5	0/10	Invalid	12.4	8000
alu3	27	0/10	12700	579	1/10	11250	248.9	8000
dyn6	30	0/10	9000	650	0/10	Invalid	26	9000
dyn7	35	2/10	11200	3141	1/10	11850	252.5	10000
alu4	36	3/10	14100	1549.4	1/10	Invalid	77.8	15000
dyn8	40	1/10	11850	1863.5	1/10	Invalid	438.9	12000
dyn9	45	1/10	13800	3684.1	2/10	Invalid	708.1	13000
alu5	45	2/10	17750	1626.4	1/10	Invalid	1092	18000
dyn10	50	2/10	16850	6452.2	2/10	Invalid	875	15000

Table 6: Multicomponent Synthesis: With vs Without Scheduling

vironment, MSS (Multicomponent Synthesis System) [10]. MSS takes as input a multi process VHDL behavior, a parameterized component library, a package library, and an overall cost constraint on the design and generates a hierarchical RTL design while simultaneously constructing a physical package hierarchy for the design.

We presented results to evaluate the performance of the approach with respect to the quality of designs produced and execution times for a number of design examples. Hierarchical RTL partitioning and package design yields good results for examples where the number of RTL components in the synthesized design are less than 200. When partitioning at the RTL netlist level, the design architecture is frozen (during high level synthesis). Alternate multichip designs cannot be explored during hierarchical RTL partitioning, whereas MSS explores the design space by considering alternate implementations during high level synthesis. Also, thermal profiling of RTL designs is too time consuming and is not viable for large designs. For almost all the examples, MSS produces better results and executes much faster than the hierarchical RTL partitioning. For smaller designs, scheduling overhead can be reduced through approximate estimation procedures to evaluate the cost of level 1 segments form individual process costs. From the results, we infer that the hierarchical behavioral partitioning is both a suitable and a viable approach to multicomponent synthesis and hierarchical packaging.

References

[1] M.C. McFarland, A.C. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," *Proc. 25th Design Automation Conference,* pp. 330–336, June 1988.

[2] M.C. McFarland, A.C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proc. of the IEEE,* Vol. 78, No. 2, pp. 301–318, Feb. 1990.

[3] R. Camposano, "From Behavior to Structure: High-Level Synthesis," *IEEE Design & Test of Computers,* pp. 8–19, Oct. 1990.

[4] K. Kucukcakar, "System-Level Synthesis Techniques With Emphasis on Partitioning and Design Planning," *Ph.D. Dissertation,* Dept. of Electrical Engineering-Systems, University of Southern California, CA, Oct. 1991.

[5] F. Vahid and D.D. Gajski, "Specification Partitioning for System Design," *Proc. 29th Design Automation Conference,* pp. 219–224, June 1992.

[6] R. Gupta and G. De Micheli, "Partitioning of Functional Models of Synchronous Digital Systems," *Proc. ICCAD-90,* Santa Clara, pp. 216–219, Nov. 1990.

[7] Nand Kumar, Srinivas Katkoori, Leo Rader and Ranga Vemuri, "Profile-Driven Behavioral Synthesis for Low Power VLSI Systems", *IEEE Design & Test of Computers,* pp. 70-84, Fall 1995.

[8] R. Dutta, "Distributed Design-Space Exploration for High-Level Synthesis Systems," *Master's Thesis,* Dept. of Electrical and Computer Engineering, University of Cincinnati, OH, 1991.

[9] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Trans. Computer-Aided Design,* Vol. 8, No. 6, pp. 661–679, June 1989.

[10] R. Vemuri et al, "An Integrated Multicomponent Synthesis Environment for Multichip Modules," *Computer,* pp. 62–74, April 1993.

[11] C.M. Fiduccia and R.M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proc. 19th Design Automation Conference,* pp. 175–181, June 1982.

[12] J. Roy, N. Kumar, R. Dutta, and R. Vemuri, "DSS: A Distributed High-Level Synthesis System," *IEEE Design & Test of Computers,* pp. 18–32, June 1992.

[13] J. Roy, "Parallel Algorithms for High-Level Synthesis," *Ph.D. Dissertation,* Dept. of Electrical and Computer Engineering, University of Cincinnati, OH, Feb. 1993.

[14] N. Kumar, "High Level VLSI Synthesis for Multichip Designs" *Ph.D. Dissertation,* Dept. of Electrical and Computer Engineering, University of Cincinnati, OH, Oct. 1994.

[15] R. Dutta, J. Roy, and R. Vemuri, "Distributed Design-Space Exploration for High-Level Synthesis Systems," *Proc. 29th Design Automation Conference,* pp. 644–650, June 1992.

[16] N. Narasimhan, J. Roy, and R. Vemuri, "Synchronous Controller Models for Synthesis from Communicating VHDL Processes," *Proc. Ninth International Conference on VLSI Design,* pp. 198-204, Jan. 1996.

APPENDIX K:

Resource Constrained RTL Partitioning for Synthesis of Multi-FPGA Designs*

Madhavi Vootukuru, Ranga Vemuri and Nand Kumar †
Laboratory for Digital Design Environments
Department of Electrical and Computer Engineering and Computer Science
University Of Cincinnati, ML 30
Cincinnati, OH, 45221-0030.
mvootuku@ece.uc.edu, ranga.vemuri@ece.uc.edu, nkumar@triquest-da.com
Ph : 513-556-4784

*This work is being done at the University of Cincinnati and is supported in part by the ARPA RASSP program and is monitored by Wright Patterson Air Force base under contract no. F33615-93-C-1316 and by the Solid State Electronics Directorate of the Wright Laboratory of the US Air Force under contract no. F33615-91-C-1811.

†Nand Kumar is currently Vice-President of synthesis, at Triquest DA Inc.

Resource Constrained RTL Partitioning for Synthesis of Multi-FPGA Designs

Abstract

In this paper we address the problem of partitioning register level designs for implementation on multiple FPGAs. The partitioner uses a modified multi-way Fiduccia-Mattheyses (FM) algorithm. Cost estimation functions needed by the partitioner to estimate the resources needed by the design on a FPGA have been developed. The methodology for estimation of resources on an FPGA device, and partitioning of the design are discussed in detail. For this paper, we use Xilinx XC4000 family of FPGAs as our target architecture. Within this family, heterogeneous selection of FPGA devices can be used.

1 Introduction

A design that has to be implemented on a Field Programmable Gate Array (FPGA) needs certain resources on the FPGA device. The kind of resources on the chip depend on the target architecture. These resources include the Configurable Logic Blocks (CLBs) containing the Function Generators (FG) and Flip-Flops (FF) for Xilinx architecture of FPGAs. If the design which has to be down-loaded onto an FPGA needs more resources than available on one device, there is a need to partition the design into multiple segments such that each of the partition segments satisfies resource constraints on the devices available. To achieve this goal, we formulate the **Multi-FPGA partitioning problem** for Register transfer level (RTL) designs as follows:

> Given a register level design represented as a net-list of components and constraints in terms of maximum number of available CLBs, function generators, flip flops and allowable user I/O pins on each chip, partition the design into a set of interconnected design segments, each of which satisfies the constraints.

The partitioning system creates one or more bit map files depending on the specified constraints. Each bit-map file can be down-loaded onto a Xilinx xc4000 family FPGA. Input to the system is a register level design which consists of a data-path and a controller. The data-path consists of a collection of components selected from a known parameterized component library. This library has various components such as adders, subtractors, multipliers, dividers, latches, multiplexers etc. The controller is specified as an algorithmic behavioral description of a finite state machine. These components are further discussed in detail in later sections of the paper.

2 Integration of tools for Synthesis and partitioning of FPGA designs

We use a high level synthesis system which takes behavioral descriptions as input and produces register transfer level descriptions of the same design. The high level synthesis system is called Distributed Synthesis System (DSS), developed at The University of Cincinnati [?], for producing RTL designs. The system produces register transfer design in two parts, namely, the 'data-path' and the 'controller'. The data-path is represented as a net-list of register transfer level components. The controller is represented as a finite state machine.

The input to the multi-FPGA partitioning system is a register level design (output of DSS) and the constraints are the FPGAs available, the library of RTL components, and the resource utilizations of all register level components

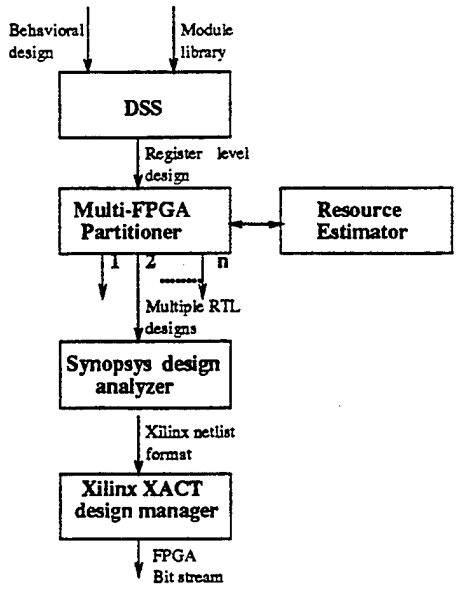


Figure 1: **Multi-FPGA synthesis flow**

for varying generic parameters. *Resource estimator* and the *partitioning algorithm* form the central components of the partitioning process. Resource estimation involves accurate estimation of necessary resources for the design and the partitioning involves the proper choice of design segments which satisfy user specified constraints. The resources here refer to the number of CLBs (packed CLBs), the number of function generators, and the number of flip-flops. Pin constraints are also taken into account while determining the partitions. If the design cannot be partitioned into the available number of chips, each with allowed number of I/O pins, the partitioner returns the best possible solution obtained during the specified number of iterations on the K-way FM partitioning algorithm.

The resource estimator works on the data-path and the controller separately and gives estimates for the overall design using these individual estimates. Once the estimation is done, it is determined whether the given design needs to be partitioned or not depending on the resources needed by the design and the specified selection of FPGA devices. The partitioner is invoked if needed. It uses a modified multi-way Fiduccia-Mattheyses algorithm [?], discussed later, to produce partition segments which satisfy the constraints. These partitions are used as input to logic synthesis tools to generate bit-map files. The design flow for obtaining programmed bit-map files for FPGAs is shown in Figure 1. We use the Synopsys design analyzer for logic synthesis of partitioned RTL designs. This produces a gate level net-list of the design in terms of hard macros and function generators from Xilinx library. Since our target implementation is Xilinx FPGA devices, we use Xilinx XDM tools for generating layouts and producing bit-map files necessary to down-load the design onto the FPGA.

3 RTL component library

The data-path part of the register level design contains components selected from a RTL component library. These components in turn use hard macros from Xilinx XC4000 family. The descriptions of these components were initially written at behavioral level and ideally, the synthesis tools should be able to understand all of the

Component	Xilinx Hard Macros used
Const_reg	Buf
Adder	Add1, Vcc, Inv
Subtractor	Add1, Gnd, Inv
Comparator	Nor2, And2b1,And2, Inv, And3b1,Xnor2
Latch	FDCE
Multiplexer	M2_1
Shift_reg	And2, Or2, Or3
Signal	And2, And3, And4, Inv, FDPE, Or2, Xnor2, FDCE
And	And2
Or	Or2
Nor	Nor2
Xor	Xor2
Xnor	Xnor2
Not	Inv

Table 1: **RT level Components in component library instantiated in RT level code**

currently available target architectures and synthesize the descriptions for a particular target architecture. In this process, the synthesis tool might produce a gate-level design, which when taken to layout might be violating the area constraints, or might be so computationally intensive that it takes several hours to synthesize. To overcome these problems, the register level components used in our library instantiate the Xilinx library components directly and thus are targeted for Xilinx xc4000 family of FPGA family. These components are parameterized for varying values of bit-width. Apart from this, components like Multiplexer are parameterized for other generics like number of inputs and number of select lines. Table 1 shows the components and the corresponding Xilinx hard macros used.

Each library module is characterized for the number of CLBs, function generators and flip-flops for different values of generic parameters. This characterized data is made available to the partitioning tool in the format shown in Table 2. This data was obtained experimentally by synthesizing several instances of each of the components with varying generic parameter values and generating the Xilinx LCA (Logic Cell Array) files. In this table, FF, FG and CLB denote the necessary number of flip-flops, function generators and CLBs respectively for each component. Each entry in this table is of the form (x,y) where 'x' is the bit-width of the component and 'y' is the resource needed. Note that the Table 2 shows only a small selection of the data for our library. For example, the resource utilizations for the multiplexer module are for 2 inputs and 1 select line.

4 Resource Estimator

Estimation functions for estimating the number of function generators, flip-flops and CLBs needed for an input design have been developed. Estimation of resources needed by a design represented as a data path and a

controller can be done by considering each of these entities separately.

Estimates for data-path : We estimate the number of function generators and flip-flops needed by the data path and use this data in determining the number of CLBs needed by the whole design when it is taken to layout. To estimate the number of function generators and flip flops, we add up these values for all the instantiated components. The generic values used to instantiate various register transfer level components, and their respective resource utilizations are obtained through table-lookup from the system database (Table 2). Logic trimming done at gate level is taken into account by reading the input net-list, and determining the signals not being used. In other words, we determine the load-less signals, if any, in the design. This does not happen frequently in the case of synthesized designs. However, for modules such as the "signal" module in Table 2, which contains multiple flip-flops, outputs of some flip-flops may not be used. The flip-flops FDPE and FDCE are the hard macros used in Xilinx FPGAs to store the bits in the clocked components. Once the load-less signals are determined, the corresponding number of flip-flops used to store these signals is subtracted from the number obtained by summing up the individual component flip-flop counts in the design. This gives the number of flip-flops necessary for the data-path. A similar procedure is followed for obtaining an estimate of function generators used by the data-path.

- No. of Flip-Flops needed by data-path (FF_{dp})= $\sum_{r \in dp} FF_count(r) - \sum_{s \in L} UFF_count(s)$

  where, $FF_count(r)$ is the number of Flip-flops of individual register level components, $UFF_count(s)$ is the unused flipflop count of component whose output signal is 's' and L is the set of load-less (unconnected) signals in the data-path.

- No. of Function Generators needed by the data-path (FG_{dp})= $\sum_{r \in dp} FG_count(r) - \sum_{s \in L} UFG_count(s)$

  where, $FG_count(r)$ is the number of Function generators of individual register level components in data-path, $UFG_count(s)$ is the number of unused function generators of component whose output signal is 's' and L is the set of load-less signals in the data-path.

Since each CLB in XC4000 family of FPGAs has 2 function generators and 2 flip-flops, the number of packed CLBs needed is determined to be half the number of flip-flops (function generators) for designs with dominating sequential (combinational) logic, that is, dominating number of flip-flops (function generators). That is,

No. of Packed CLBs needed by the data-path = 0.5 * Max(FF_{dp}, FG_{dp})

Estimates for controller : The necessary number of function generators and flip-flops in the controller part of the design can be estimated by studying the description of the finite state machine (FSM). The number of states in the controller is the main factor which determines the amount of logic required on the chip. The number of state variables depends on the number of states in the FSM and is given by log_2(number of states). A register whose bit-width is equal to the number of state variables is needed to store the present state and next state variables.

The elaborated FSM is represented as a set of multiplexers and gates by the logic synthesis tool. The size of inputs and select signals to the multiplexers was found to be proportional to the control word length, and number

S.No.	Module Name	(Bit-width, Resource count)
1	Latch	FF : (1,1),(2,2),(4,4),(8,8),(16,16) FG : (1,0),(2,0),(4,0),(8,0),(16,0) CLB : (1,1),(2,1),(4,2),(8,4),(16,8)
2	Multiplexer	FF : (1,0),(2,0),(4,0),(8,0),(16,0) FG : (1,1),(2,2),(4,4),(8,8),(16,32) CLB : (1,1),(2,1),(4,2),(8,4),(16,16)
3	Signal	FF : (1,7),(2,12),(4,19),(8,35),(16,67),(32,80) FG : (1,3),(2,5),(4,7),(8,13),(16,25),(32,40) CLB : (1,4),(2,6),(4,10),(8,17),(16,34),(32,40)
4	Comparator	FF : (1,0),(2,0),(4,0),(8,0),(16,0),(32,0) FG : (1,3),(2,8),(4,18),(8,37),(16,38),(24,60),(32,157) CLB : (1,1),(2,4),(4,9),(8,18),(16,19),(24,30),(32,78)
5	And	FF : (1,0),(2,0),(4,0),(8,0),(16,0) (32,0),(64,0) FG : (1,1), (2,2), (4,4), (8,8), (16,16) (32,32), (64,64) CLB : (1,1), (2,1), (4,2), (8,4), (16,8) (32,16), (64,32)
6	Or	FF : (1,0),(2,0),(4,0),(8,0),(16,0) (32,0),(64,0) FG : (1,1), (2,2), (4,4), (8,8), (16,16) (32,32), (64,64) CLB : (1,1), (2,1), (4,2), (8,4), (16,8) (32,16), (64,32)
7	Nor	FF : (1,0),(2,0),(4,0),(8,0),(16,0) (32,0),(64,0) FG : (1,1), (2,2), (4,4), (8,8), (16,16),(32,32), (64,64) CLB : (1,1), (2,1), (4,2), (8,4), (16,8) (32,16), (64,32)
8	Xor	FF : (1,0),(2,0),(4,0),(8,0),(16,0),(32,0),(64,0) FG : (1,1), (2,2), (4,4), (8,8), (16,16),(32,32), (64,64) CLB : (1,1), (2,1), (4,2), (8,4), (16,8),(32,16), (64,32)
9	Xnor	FF : (1,0),(2,0),(4,0),(8,0),(16,0),(32,0),(64,0) FG : (1,1), (2,2), (4,4), (8,8), (16,16) (32,32), (64,64) CLB : (1,1), (2,1), (4,2), (8,4), (16,8),(32,16), (64,32)
10	Const_reg	FF : (1,0), (2,0), (3,0), (4,0), (5,0), (6,0), (7,0), (8,0), (16,0) FG : (1,0), (2,0), (3,0), (4,0), (5,0), (6,0), (7,0), (8,0), (16,0) CLB : (1,0), (2,0), (3,0), (4,0), (5,0), (6,0), (7,0), (8,0), (16,0)
11	Adder	FF : (1,0),(2,0),(4,0),(8,0),(16,0),(32,0),(64,0) FG : (1,1),(2,3),(4,6),(8,12),(12,18),(16,24),(20,30),(32,48),(64,64) CLB : (1,1),(2,1),(4,3),(8,6),(12,9),(16,12),(20,15),(32,24),(64,32)
12	Subtractor	FF : (1,0),(2,0),(4,0),(8,0),(16,0),(32,0),(64,0) FG : (1,1),(2,3),(4,6),(8,12),(12,18),(16,24),(32,48),(64,64) CLB : (1,1),(2,1),(4,3),(8,6),(12,9),(16,12),(20,15),(32,24),(64,32)
13	Shift_Reg	FF : (1,3),(2,4),(3,6),(4,8),(5,10),(6,12),(7,14),(8,16)(16,32),(32,40) FG : (1,1),(2,4),(3,6),(4,8),(5,10),(6,12),(7,14),(8,16) (16,32),(32,40) CLB : (1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,7) (16,16),(32,20)
14	Not	FF : (1,0),(2,0), (3,0),(4,0), (5,0),(6,0),(7,0),(8,0), (16,0),(32,0) FG : (1,0), (2,0), (3,0),(4,0), (5,0),(6,0),(7,0),(8,0), (16,0), (32,0) CLB : (1,0), (2,0), (3,0),(4,0), (5,0),(6,0),(7,0),(8,0), (16,0), (32,0)

Table 2: **Data provided in Component-data file (input to estimator)**

of function generators was found to be proportional to the number of states in the FSM. We conducted a series of experiments and found that the number of function generators needed by the FSM is lesser if the control word in the FSM depends only on the current state (Moore machine) rather than on the current state and the control inputs (Mealy machine). The number of gates (and hence the number of function generators) needed by the FSM depends on the number of nested 'case' statements in the FSM description in VHDL. This implies that every time the input flags or input state bits are checked for assigning a value to the output of FSM, the number of gates/function generators increase. Using a large number of designs, the increase was found to be 3 function generators for each nested 'case' statement. Hence the factor 3 in the equation below. The length of the control word, which is the output of the FSM does not significantly affect the amount of logic necessary for the controller. On the other hand, number of states in the controller has a major influence on the resources needed on an FPGA. We found that there is almost an exponential increase in the number of function generators necessary with increasing states in a controller. This is due to the extra logic that is needed for assigning values to the signals for each state that is included in the finite state machine. The exponent S was determined to be 2.0. This was found by varying the number of states in the controller, keeping all the other factors constant and producing LCA of the FSM.

- No. of flip-flops needed by controller (FF_c) = No. of state variables in the FSM.
- No. of function generators needed by the controller (FG_c) =

  No. of state variables ** S +
  No. of bits in control word * C +
  No. of nested 'case' statements * F

  where, S = 2.0, C = 0.3 and F =3.

  Since the number of function generators in a FSM is usually much larger number than the number of state variables (number of flip-flops) in the FSM, the number of packed CLBs needed by the controller is estimated to be half of the number of function generators.
- No. of Packed CLBs needed by the controller = 0.5 * Max(FF_c, FG_c)

Estimates for the complete design : Estimates of resources needed by data-path and controller can be used in determining the number of function generators, flip-flops and packed CLBs needed by the complete design.

- The number of function generators in the complete design (FG_{RTL}) = (FG_{dp} +FG_c). For tighter estimates, a multiplication factor G can be taken into account, where **G** represents the global optimization factor.

  By synthesizing and analyzing a large number of designs, we found that the global optimization factor is found to lie between 0.6 and 0.9 depending on the amounts of combinational and sequential logic involved in the design.
- Number of flip-flops in the complete design (FF_{RTL})= FF_{dp} + FF_c
- Packed CLB count for the complete design (CLB_{RTL}) = 0.5 * Max(FG_{RTL}, FF_{RTL})

5 Partitioning Algorithm For Producing Multiple FPGA designs

Partitioning of a design involves determining the constraints such as the overall resource utilizations on an FPGA, and constructing the partitions subject to these constraints.

The partitioner takes the input RTL net-list and produces multiple RTL design segments. Each segment is subject to the following constraints:

1. *Resource Constraint:* The resources required by any segment of the design should not exceed the maximum allowed values set by the user on the particular FPGA part number. The constraints here refer to number of CLBs, function generators and flip-flops present in the FPGA device made available to the partitioning system. Let these constraints be denoted by CLB_s, FG_s and FF_s, where 's' denotes an FPGA device available.

2. *Pin Constraint:* Because of the limitation on the number of user I/O pins on any FPGA chip, we partition the input design into segments in such a way that the interconnect between the segments does not need more I/O pins than available. In other words, the number of pins on any segment should not exceed the allowed number of user I/O pins P_s on each chip. This is checked for all the partitions of the design.

3. *Overall design constraint:* Number of segments after partitioning should not exceed the allowed number of FPGA devices.

We use the modified multi-way Fiduccia-Mattheyses algorithm [?] for partitioning an input design into multiple design segments. The multi-way FM partitioning algorithm used is shown in Figure 2. The partitioner begins by reading the package library. This package has the information about the FPGA devices available in the format shown in Table 6. The number of partitions is initialized to 1, and the FM partitioner is invoked with the number of partitions and the package library. The FM partitioner in turn invokes the K-Way Fiduccia-Mattheyses algorithm which works on the input design (in the form of a Graph 'G'), the number of required partitions and the FPGA package. It returns a Result, which is a flag to indicate whether or not all the partition segments are assigned a device. In the event when the Result is false (that is, not all partition segments have a fitting chip amongst the devices made available by the user), the partitioner increments the number of required partitions and repeats the above process while this number does not exceed the total number of available devices, or till a successful mapping of partition segments to FPGA devices is found.

The K-Way partitioner initializes the values of total number of CLBs, function generators and flip-flops for the whole design, which are obtained as output of the *estimation* functions. It then determines the minimum number of FPGAs needed by the design. This is calculated as,

Minimum number of partition segments = $\lceil Packed\ CLBs\ for\ complete\ design\ /\ MaxCLB \rceil$

where, MaxCLB = Number of CLBs available on the largest FPGA device available.

Once the number of chips is determined, a random initial partition of N partition segments is created by the K-Way FM algorithm, where N is the minimum number of chips. As a result, the graph G of V vertices is partitioned into N segments, each with a fixed number of nodes. The initial partition is saved as *Best* partition. The pins on all partitions are calculated by compute_pins_on_all_partitions() and the value saved as *best_pins*. K-Way partitioning is carried out by repeatedly invoking the standard FM bi-partitioning algorithm [?] on pairs of

```
Multi_way_FM_Partitioner()
begin
 Package_ptr ← Read_package_data()
 Num_of_partitions =1
    While (Num_of_partitions <= Available_num_of_chips)
      Result ← FM(Num_of_partitions, Package_ptr)
         if (Result =1) then
           return partitions
         else
          Num_of_partitions ←  Num_of_partitions + 1
     end while
     if (Result = 0) then
       return Best possible partition and prompt user for bigger FPGA devices
     end if
end


int FM(Num_of_partitions, Package_ptr)
begin
   Initialize_private_data()
   K-way(G,  Num_of_partitions, Package_ptr)
   Result ← Check_assigned_chips()
   return Result
end
```

Figure 2: **Algorithm for Multi-Way FM partitioning**

partition segments. two_way_fm() tries to improve bi-partitions by moving one node at a time from one partition to the other. Each time a move is made, *check_chip_constraint(S)* is invoked to ensure that each partition segment satisfies the constraints. This function checks if the partition segment S satisfies the constraints such as CLBs, function generators, flip-flops and pins available on the chip and returns the *status* to K-Way FM. The status is true if the constraints are met by the partition segment and false otherwise. Once a chip is found in which the partition segment fits in, the device part number is assigned to this partition segment.

The K-WAY FM algorithm is invoked repeatedly until either (1) a solution that satisfies the specified constraints is found, or (2) a user specified limit on number of iterations (MAX_ITER_CNT) is exceeded. The partitioner returns either a set of partition segments that satisfy the constraints or the best possible solution (if the constraints could not be met by all the partitions).

```
K-WAY(G, Num_of_partitions, package_ptr)
begin
  Best ← initialize()
  Min_chips ← calculate_min_chips()
  if Min_chips > Num_of_partitions then
    N ← Min_chips
  else N ← Num_of_partitions
    Create_initial_partition()
    Compute_pins_on_all_partitions()
    best_pins ← Pins(Best)
    S ← null
    continue_part ← TRUE
    iteration_cnt ← 1
    improve_cnt ← 1
    while continue_part = TRUE do
     for i = 1 to n−1 do
        for j = i+1 to n do
           two_way_fm(s_i, s_j)
        end for
     end for
     curr_pins ← Pins(S)
     iteration_cnt ← iteration_cnt + 1
     status ← check_chip_constraint(S)
     if curr_pins < best_pins ∨ status = TRUE then
        improve_cnt ← 1
        Best ← S
     else
        improve_cnt ← improve_cnt + 1
     end if
     if  iteration_cnt = MAX_ITER_CNT ∨
        improve_cnt = MAX_IMP
       then   cont_part ← FALSE
     end if
    end while
  return(Best)          /* retrieve best partition */
end
```

Figure 3: **Algorithm for partitioning (Contd.)**

```
Check_chip_constraint(S)
begin
  status ← TRUE
  for all s_i ∈ S do          /* segments in partition */
     if CLBs(s_i) > CLB_s ∨ pins(s_i) > P_s ∨
      FG(s_i) > FG_s ∨ FF(s_i) > FF_s
     then  status ← FALSE
     end if
  end for
  return(status)
end
```

Figure 4: **Algorithm for partitioning (Contd.)**

6 Implementation and results

We have developed a vertically integrated system for a top-down design flow for FPGA synthesis with the high level synthesis system, DSS as the front end and multi-way Fiduccia-Mattheyses algorithm being used for producing partitions of the input design. We used Xilinx XC4000 as our target FPGA family and used Synopsys design analyzer and Xilinx XACT design manager tools, to produce gate-level net-list and programmed bit map files respectively for FPGA implementation. We developed estimation procedures for estimating the resources needed by a register level design to be implemented on the FPGA devices. We used these estimates and produced multiple register level designs using the Multi-Way FM partitioning algorithm.

Tables 3, 4 and 5 show actual and estimated resources needed by the data-path, controller and the complete design respectively. It can be observed from these tables that the estimated and actual values of flip-flops match exactly for the data-path and controller since the correct utilizations for the RTL components is provided to the estimator (in the case of data-path) and the number of flip-flops can be correctly known from the number of states in the FSM (in the case of controller) . Since the number of flip-flops for the overall design are calculated from those needed by the data-path and controller, the estimated values of flip-flops needed by the overall design match exactly with the actual values. On the other hand, we find that estimated and actual values of function generators in the data-path, controller and overall design differ on an average by about 6% in the case of data-path, 11% in the case of controller and 9% for the complete design. The number of packed CLBs for the complete design differ from the estimates due to the discrepancies in the estimates of function generators, which is in turn due to the FSM synthesis methodology used by the logic synthesis tool and global optimization over function generators . This deviation from the actual values was found to be about 10% on an average.

Table 6 shows a sample FPGA device selection provided by the user. This has information such as the FPGA part number, number of chips of each kind available, and the resources available on each chip. Constraints and corresponding partitions obtained from the partitioner for a number of designs are shown in Table 7. The design utilizations in this table refer to the estimated values of resources needed by each of the designs. For the first example, the design fits in only one device and hence the mapping is as shown. In the case of 'DCT' example, suitable partitions for the devices available cannot be found. Hence there are no partitions available. Instead the partitining system prompts the user to try with bigger chips. The execution time of the partitioning tool for

these examples lies between 1.2 sec to 4.0 sec.

When a design is partitioned into multiple design units, the delay on the nets passing from one unit to the other might be so large that the the frequency of operation of the overall design is drastically reduced. We are currently extending our partitioning engine to incorporate delay constraints.

Design	No. of RTL comp's in the data-path	Function generators		Flip-flops	
		Estimate	Actual	Estimate	Actual
TLC	33	47	44	48	48
SS-prod	34	423	374	369	369
DCT	23	157	187	209	209
Find	57	350	384	184	184

Table 3: **Estimated and actual values for data-path**

Design	Num of states	Control word length	Function generators		Flip-flops	
			Estimate	Actual	Estimate	Actual
TLC	34	40	109	86	6	6
SS-prod	37	40	143	132	6	6
DCT	38	30	129	99	6	6
Find	76	70	182	199	7	7

Table 4: **Estimated and actual values for controller**

Design	Function generators		Flip-flops		Packed CLBs	
	Estimate	Actual	Estimate	Actual	Estimate	Actual
TLC	140	156	54	54	70	78
SS-prod	510	411	375	375	255	205
DCT	258	312	215	215	129	156
Find	479	520	191	191	239	260

Table 5: **Estimated and actual values for complete design**

FPGA Part	CLBs	Function Generators	Flip-Flops	I/O pins	Num available
XC4002	64	128	128	64	1
XC4003	100	200	200	80	2
XC4005	196	392	392	112	1
XC4010	400	800	800	160	0

Table 6: **Sample FPGA Device Selection.**
Note : 'Number available' is specified by the user. The rest of the data is provided by a configuration file to the partitioning tool.

Acknowledgments

We thank Narendra Narasimhan, Vinoo Srinivasan and Sriram Govindarajan, University of Cincinnati for their help and advice. In particular, we thank Vinoo Srinivasan and Sriram Govindarajan for their contribution in the development of the RTL library targeted for FPGAs.

Design	Design utilization	FPGA package data Part(No. available)	Partitions			
			Chip1	Chip2	Chip3	Chip4
TLC	CLBs=70(78), FGs=140(156), FFs=54(54)	XC4002 (2) XC4003 (1)	XC4003 FGs=140(156) FFs=54(54)	- - -	- - -	- - -
TLC	CLBs=70(78), FGs=140(156), FFs=54(54)	XC4003 (2) XC4004 (1)	XC4003 FGs=140(156) FFs=54(54)	- - -	- - -	- - -
SS-Prod	CLBs=255(205), FGs=510(411), FFs=375(375)	XC4004 (1) XC4005 (1)	XC4004 FGs=204(152) FFs=168(168)	XC4005 FGs=306(259) FFs=207(207)	- - -	- - -
SS-Prod	CLBs=255(205), FGs=510(411), FFs=375(375)	XC4005 (2)	XC4005 FGs=204(152) FFs=168(168)	XC4005 FGs=306(259) FFs=207(207)	- - -	- - -
DCT	CLBs=129(156), FGs=258(312), FFs=215(215)	XC4003 (2)	- - -	- - -	- - -	- - -
DCT	CLBs=129(156), FGs=258(312), FFs=215(215)	XC4004 (1) XC4005 (2)	XC4004 FGs=258(312) FFs=215(215)	- - -	- - -	- - -
Find	CLBs=239(260) FGs=479(520) FFs=191(191)	XC4008 (1)	XC4008 FGs=479(520) FFs=191(191)	- - -	- - -	- - -
Find	CLBs=239(260) FGs=479(520) FFs=191(191)	XC4004 (2) XC4005 (2)	XC4004 FGs=50(57) FFs=51(51)	XC4004 FGs=178(203) FFs=23(23)	XC4005 FGs=168(173) FFs=84(84)	XC4005 FGs=83(87) FFs=33(33)

Table 7: **Constraints and results of Partitioner**

Note : The resources are represented as estimated value (actual value).

References

[1] Jay Roy, Nand Kumar, Rajiv Dutta and Ranga Vemuri, "DSS : A Distributed High-Level Synthesis System", IEEE D&T Of Computers, vol. 9, No. 2, June 1992.

[2] Nand Kumar, "High Level VLSI Synthesis For Multichip Designs", PhD Dissertation, University of Cincinnati. October 1994.

[3] C.M.Fiduccia and R.M.Mattheyses, "A Linear-Time Heuristic for Improving Network partitions," Proc. 19th Design Automation Conference, pp. 175-181, June 1982.

[4] Jay Roy, Rajiv Dutta and Ranga Vemuri, "Distributed Design Space exploration For High Level Synthesis Systems", 29 th ACM/IEEE Design Automation Conference, June 1992.

[5] Daniel D. Gajski, Loganath Ramachandran, "Introduction To High Level Synthesis", IEEE D&T Of Computers, 1994.

[6] A. Sangiovani-Vincentelli, "Synthesis Methods For Field Programmable Gate Arrays", IEEE Proceed ings, July 1993.

[7] Nam-Sung-Woo, Jaesook Kim, "Efficient Method For Partitioning Circuits for Multiple FPGA Implementation", 30 th Design Automation Conference, June 1993.

[8] R. Camposano, "From Behavior to Structure: High-Level Synthesis", IEEE Design & Test of Computers, pp. 8-19, Oct. 1990.

[9] R.Rajaraman, D.F.Wong, "Optimum Clustering for Delay Minimization", IEEE Transactions on CAD, Vol 14, pp.1490-1495, Dec 1995.

[10] R. Camposano and W. Wolf, "High-Level VLSI Synthesis", Kluwer Academic Publishers, Boston, 1991.

[11] XILINX XC4000 FPGA data Book, XILINX Inc., 1994.

[12] XILINX Unified Libraries, XACT Libraries guide, April 1994.

APPENDIX L:

Using Declarative Specifications and Case-Based Planning for System Synthesis*

Perry Alexander, Philip Baraona, and John Penix
Knowledge-Based Software Engineering Lab
Department of Electrical and Computer Engineering
The University of Cincinnati
Cincinnati, OH 45221-0030
Perry.Alexander@UC.edu,{pbaraona,jpenix}@thor.ece.uc.edu

*Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615-93-C-1316.

Keywords: declarative specification, case-based reasoning, system synthesis, two-tiered specification.

Abstract

Synthesis of pragmatic systems from high-level specifications requires representation and application of both functional requirements and constraints. This work presents a language for representing requirements and constraints in VHDL design representations and a prototype case-based synthesis system. VSPEC is an annotation language for VHDL developed to support axiomatic representation of requirements for system synthesis. VSPEC descriptions serve as synthesis goals and verification criteria. A prototype case-based synthesis system is also presented that uses VSPEC requirements as goal statements and descriptions of potential solutions. This prototype system demonstrates how synthesis can be performed at the systems level and how constraints can be used to implement a simple concurrent engineering process.

1 Introduction

VSPEC is motivated by the need to specify digital system requirements in an implementation independent fashion. Qualitatively, system requirements specify "what" a system should achieve without specifying "how" it should be done. Design specifications are developed from requirements and describe "how" requirements are implemented. Although VHDL [14] supports specification of specific designs, it does little to support requirements specification. In addition, VHDL does not support a consistent representation of constraints. Thus, requirements specification in VHDL and systems level synthesis from VHDL specifications are not practical activities.

Lack of requirement and constraint specification has little effect when designing systems requiring few levels of abstraction. Excellent VHDL synthesis tools exist at the RTL level. However, there is a growing need for systematic design of very large, abstractly defined systems. Without constraint information and precise requirements definition, effective systems engineering and concurrent engineering are impossible, and automated synthesis is even more difficult than this.

With requirements and constraints specified, some degree of systems level design synthesis is possible. The case-based synthesis system presented here demonstrates how constraints can be integrated into an automated design process. System synthesis occurs in a typical, function oriented manner. However, constraints help rank potential solutions during case retrieval and are verified at each level of abstraction as the design progresses.

This work concentrates on two subjects: the VSPEC language and a prototype synthesis system. First, the VSPEC interface language is presented. The general syntax and notations used by VHDL and VSPEC are discussed followed by an example specification. Specific attention is given to describing how VSPEC represents both functional and non-functional system requirements. Also discussed is the relationship between algebraic specification and VSPEC with a presentation of syn-

thesis goal derivation. Second, a case-based reasoning synthesis process is presented. The basics of the technique are explained via an annotated example which highlights the role of constraints in the synthesis process. To conclude, perceived limitations of the synthesis system and our current research directions are described.

2 Design and Requirements Specification

Three basic constructs are used to specify a design in VHDL: (1) the `entity` specifies the interface of a system; (2) the `architecture` specifies the behavior and/or structure of a system; and (3) the `configuration` associates a specific `architecture` with an `entity`. The designer specifies a device interface using the `entity` construct, develops one or more behavioral or structural descriptions using the `architecture` and selects a specific implementation for the `entity` using the `configuration` construct.

Each `architecture` represents a potential design at some level of abstraction. Behavioral specifications describe the behavior of a solution using an Ada-like programming language. Structural specifications indicate how components are composed to construct a solution. In both cases, specific candidate designs are represented.

Representation of system requirements in VHDL is restricted to an operational style - a "program" is written that describes an artifact having desired characteristics. Although the operational style is an excellent means for describing specific designs, it is not well-suited for describing system requirements for several reasons:

1. It forces representation of a specific design, thus introducing implementational bias.
2. It does not adapt easily to representation of performance constraints.
3. Unimportant characteristics are indistinguishable from required characteristics of the design.

4. Users must deal with unnecessary detail.

2.1 VSPEC Requirements Specification

Figure 1a is an example VHDL `entity` representing a component that searches a collection of records for a specific record. Note there is no indication of what the component must accomplish or what performance constraints exist for it. The result is a black-box view of the component with no indication of requirements, as shown in Figure 1b. An architecture can be developed, but such an architecture exhibits the negative characteristics described above.

A solution to requirements representation in VHDL is VSPEC, a Larch interface language [8] developed for VHDL synthesis. The Larch family of specification languages consists of a collection of application specific interface languages and a common shared language. Each interface language defines sets of specification primitives containing useful constructs in a target application language. The shared language serves two purposes. First, it provides a target formal system for translating interface specifications. Second, it provides a language for writing auxiliary specifications and handbooks of common components.

The traditional shared language is a first order algebraic language call LSL. In VSPEC, the primary shared language is REFINE [1], due to its support for transformation and synthesis, its formal basis, and its potential for execution.

Figure 2a shows the VSPEC representation for the same search as the VHDL `entity` in Figure 1a. The added clauses specify input conditions, output conditions and constraints. Figure 2b shows a graphical representation of the same information. The VSPEC definition indicates that `Vcc` must be less that or equal to 5 and that the area ($x \times y$) must be less than 0.3. No constraints are placed on heat dissipation (`H`), clock speed (`Clk`) or timing.

The specification associated with Figure 2 avoids many of the problems with the operational

specification style. A search routine is specified independently of any implementation by the ensures clause. The designer need not be concerned with the details of the search algorithm at the requirements level. Only characteristics necessary for specifying a search are included. Constraints are clearly specified in the constrained by clause and do not interfere with the functional specification.

2.2 The VSPEC entity

All VSPEC annotations affect only the VHDL entity. No changes are made to architecture structures or any other VHDL structure. VSPEC clauses are grouped into four broad classes: (1) those that define a devices function; (2) those that define internal state variables; (3) those that define constraints; and (4) those that relate VHDL data structures to formal representations.

2.2.1 VSPEC Clauses and Logic

The general form of a VSPEC clause is a keyword followed by a logical sentence. The keywords indicate what requirement the logical sentence specifies. Each logical sentence is written in typed first-order predicate calculus with extensions to the logic that allow the use of sets and sequences in specifications. The logic follows the basic syntax of REFINE, the language used for system synthesis, to support easy translation and some degree of execution.

There are six basic VSPEC clauses:

- **requires** – specifies sufficient conditions on inputs and state for **entity** execution
- **ensures** – specifies necessary conditions on outputs and state following **entity** execution
- **constrained by** – specifies non-functional performance constraints
- **modifies** – specifies what the **entity** may alter
- **based on** – associates VHDL data types with REFINE definitions

- `state` -- defines a collections of variables that represent the `entity`'s internal state

VSPEC clauses may only access variables and signals defined in an `entity port`, the `state` clause or quantified in a logical expression. VSPEC is strongly typed and all variables must have an associated type, including those bound by quantifiers. Although REFINE allows type inferencing, VSPEC does not.

2.2.2 Functional Requirements

The functional requirements of a VSPEC `entity` are defined using the `requires` and `ensures` clauses. The `requires` clause specifies a logical expression, $I(x)$, that must be true for the `entity` to perform its operation. The `ensures` clause specifies necessary state conditions, $O(x, z)$, resulting from `entity` execution given a particular input. Formally, any `architecture` implementing an `entity` must obey the condition:

$$\forall x : D \bullet I(x) \Rightarrow O(x, F(x)) \tag{1}$$

where D is the domain of the transform $F(x)$ is the transformation performed by the `architecture`.

The `requires` Clause

The `requires` clause, $I(x)$, is a logical expression defined over all ports, signals and variables that may provide input to the transform. $I(x)$ is true when x is a valid input. $I(x)$ is a precondition for `entity` execution. When it is true, the `entity` must produce valid output.

The `ensures` Clause

The `ensures` clause, $O(x, z)$, is a logical expression defined over all ports, signals and variables. $O(x, z)$ is true when z is a valid output given x as input. $O(x, z)$ is a postcondition for `entity`

execution and states necessary conditions placed on `entity` outputs and state variables.

2.2.3 Constraints

Constraints express characteristics an `entity` must exhibit that are not a part of its function. For example, heat dissipation constraints frequently affect selection of valid designs, but heat is a side effect of the technology. It has little to do with the input and output relationships specified in the requires and ensures clauses.

Although constraints do not affect function, they are critical in hardware system design. In VSPEC there are two sources of constraint. The first is the `constrained by` clause that specifies several performance constraints common in hardware design. The second is the `modifies` clause that limits what the `entity` can alter in performing its function.

The `constrained by` Clause

The `constrained by` clause is a conjunction of predefined variables and relations with fixed values. VSPEC currently supports providing constraint information for heat dissipation, area, clock speed, power consumption and pin-to-pin timing. To specify constraint, one chooses a constraint type and uses it in a relation. For example, to specify heat dissipation less than 1 watt and power consumption less than 10 watts, the logical sentence `heat =< 1 and power =< 10` is included in the `constrained by` clause.

Timing requires a somewhat more complicated representation. Here one specifies an interval between two pins, then relates that interval to a constant time. For example, `(a<->b) =< 10` specifies that the time between a signal arriving at port `a` and port `b` producing a signal must be less than 10.

The modifies Clause

The modifies clause specifies a collection of ports, signals and variables that may be modified by the entity. The modifies clause indicates what effects and side effects are allowed. Only outputs may be specified in a modifies clause. Of particular interest is the ability to specify the direction of buffer type ports.

2.2.4 Abstract Data Types

The semantics of VHDL data types must be defined before reasoning about their properties is possible. Elemental data types such as integer and bit have definitions loaded as a part of the VSPEC system. Thus, when using a basic VHDL type, the semantics of that type are present by default.

The based on Clause

User defined data types such as arrays and records must be defined as a part of the definition process because they cannot be defined *a priori*. This is accomplished using the based on predicate. The logical expression defined in a based on clause defines the semantics of a user defined type. To support this specification process, VSPEC includes standard schemas for defining sets, sequences, arrays and tuples. These schemas are used in conjunction with parameter morphism to define associated VHDL types specific to user needs.

2.2.5 System State

The notion of system state is typically not supported directly by axiomatic specification techniques. A computation unit is defined by a transform that relates inputs to outputs. Thus, to include state in a specification it must be specified as both an input and an output of the transform. However,

specification of state-based systems is natural to hardware designers and suggesting that state representation be an input to the VHDL `entity` is not natural. Using the two-tiered specification approach, state can be managed by: (a) supporting the definition of local state variables; and (b) using state maintaining features of `port` signals. Instead of specifying a function that maps input signals defined in the `port` definition to outputs in the same `port` definition, specify a function that maps inputs and state maintaining objects to outputs and state maintaining objects.

The `state` clause

The `state` clause is a collection of variables that store state within a VSPEC `entity`. Like VHDL variables and signals, these variables maintain their values from one invocation of the `entity` to the next. All `state` variables are defined locally and are not visible outside the entity.

Ports

Variables defined an `entity`'s `port` definition may maintain their state. Variables of type `buffer` may be inputs or outputs and are not re-initialized unless a signal of some type is driving them. Variables of type `out` and `inout` also maintain their state.

2.3 Generic Architectures in VSPEC

VSPEC supports representation of high level, abstract architectures using the `architecture` construct from VHDL. No modifications or annotations are necessary - simply specify `entity` structures accessed by the `architecture` using VSPEC.

Figure 3 represents a two component architecture for solving the element search problem. The `search entity` is identical to the one in Figure 2a which serves as the starting point for designing the system. The next step is creating a VHDL `architecture` that solves the problem specified by

the VSPEC `entity`. In this example, `architecture` structure solves this problem by breaking it up into two sub-components: one which sorts the input and one which retrieves the proper element from the sorted list. This `architecture` was generated using the synthesis technique discussed in Section 3. The result of breaking the problem into two sub-components is two new VSPEC entities that describe the subcomponents. Notice that the combination of the functional and performance constraints of each sub-component meet the constraints specified by the `search entity`. The next step in the design process is to generate VHDL architectures for each of these sub-components. The `behavior architecture` is an example of a solution for the `bin_search entity`.

2.4 VSPEC and Algebraic Specification

Any VSPEC definition can be transformed into a formal definition. The form of this definition is an algebraic specification based on an extension of domain theories as defined in CYPRESS [18] and KIDS [20, 19]. The basic form of a domain theory is a tuple consisting of the function domain (D), range (R), input precondition ($I(x : D)$) and output postcondition ($O(x : D, z : R)$) commonly referred to as a *DRIO* model. The *DRIO* model for any VSPEC `entity` can be constructed using the following rules:

$D = d_1 \times d_2 \times \ldots \times d_n$ where d_k is the sort (defined by the based on clause) representing the type associated with an `in`, `inout`, or `buffer` ports, or a state variable

$R = r_1 \times r_2 \times \ldots \times r_m$ where r_j is the sort representing the type associated with an `out`, `inout`, or `buffer` port listed in the `modifies` clause, or a state variable

$I(x : D) = I_v(x : D)$ where $I_v(x : D)$ is the logical sentence defined by the `requires` clause

$O(x : D, z : R) = O_v(x : D, z : R)$ where $O_v(x : D, z : R)$ is the logical sentence defined by the `ensures` clause

Additionally, constraints must be defined as a part of the algebraic statement. The simplest means of accomplishing this is to include predicates representing constraints in the output function of the *DRIO*. However, constraints are not functional. Specifying constraints in their own clause is an attempt to separate constraint from function. Additionally, constraints in their current form do not depend on variables defined in the `entity`[1]. Thus, constraints are added to the *DRIO* model through a specification extension that adds logical representations of constraints. Effectively, the *DRIO* model becomes a *DRIOC* model.

$C(c_1 : C_1, \ldots, c_n : C_n) = C_v(c_1 : C_1, \ldots, c_n : C_n)$ where c_k is a constraint variable such as heat or area, C_k is a sort associated with a constraint variable and C_v is the logical expression defined in the `constrained by` clause

With addition of constraints, the goal of the design activity becomes finding an architecture that performs the transform $F : D \rightarrow R$ such that:

$$\forall x : D \bullet I(x) \Rightarrow O(x, F(x)) \wedge C(c_1, \ldots, c_n) \tag{2}$$

Thus, the goal of the synthesis activity is generation of a transform mapping the current state and inputs into the next state and outputs such that the output condition and constraints are satisfied.

3 System Synthesis

The case-based reasoning model used by the synthesis system is based on the standard approach of retrieval, adaptation, and evaluation. In the following sections, each of these activities is described.

[1] A more complex constraint model could certainly include variables and signals. Our current constraint model does not allow this.

The similarity metric, features and feature types are described followed by a description of the three stage retrieval process. Adaptation via rule application and by replacing case components is described next followed by a brief description of the evaluation process.[2].

Given a VSPEC specification and its *DRIOC* model equivalent, planning techniques apply to system synthesis. The general goal of planning is to accumulate a partially ordered bag of actions that achieve the end result. This goal is analogous to the design of general systems where one is searching for a collection of interconnected devices for solving a problem. Effectively, I and O define pre- and post-conditions for a component. In planning terms, this is identical to the description of a goal or plan action. Consider the goal of system synthesis described in Equation 1. This is exactly the goal of a planning system - given a pre-condition, find a sequence of actions that necessarily imply a desired post-condition.

Although any number of planning techniques apply, case-based planning is discussed here. A method derived from the ASP-II[4] analysis planner and refined in the BENTON[2, 3] is applied. The ASP-II planner used case-based reasoning to synthesize simulation actions given characteristics described in a `before` clause (pre-condition) and an `after` clause (post-condition). ASP-II supports the replacement of failed actions in a plan using a technique called adaptation by re-planning.

Adaptation by re-planning works by inferring a goal from the state change caused by a plan action. The system state is known before the action is executed from the post-condition of the preceding action and the system state after execution from the pre-condition of the following action. Thus, if an action or sequence of actions failed the goal of the action could be retrieved and used as a goal for a new planning process.

In our case-based synthesis system, VHDL components are analogous to plan actions and VHDL

[2]For a more formal description of the retrieval and adaptation processes, please refer to the BENTON case-based reasoning component [3]

architecture structures are analogous to composite plans. The *DRIOC* form of VSPEC requirements expresses exactly what a plan action does - $I(X)$ expresses a precondition and $O(x, z)$ expresses a post condition. Thus, VSPEC requirements can be used to generate goals for synthesis processes to replace components analogously to plan actions in ASP-II.

3.1 Example Problem

To demonstrate the case-based reasoning technique, synthesis of a VHDL component implementing a search system will be used as an example. Figure 4 represents the VSPEC requirements for the searching component. This requirements specification states that a list of elements and a key are input with the element associated with the key output. The `requires` clause states that no preconditions exist on the input set. (Note that the `entity port` definition assures inputs are of the correct type.) The `ensures` clause specifies that if an element in the input array has a key value associated with k, that element is returned by the function.

The VSPEC `entity` is parsed and the result is a *DRIOC* specification of the following form:

$D =$ $\quad seqence(element) \times integer$
$R =$ $\quad element$
$I(x) =$ $\quad true$
$O(x, z) =$ $\quad \forall e : element \bullet output = e \Leftrightarrow e \in input \wedge k = key(e)$
$C =$ $\quad power \leq 10$

3.2 Cases

Each stored case is a triple consisting of a problem description, feature set, and potential solution. The problem description is the *DRIOC* translation of the VSPEC requirements, the feature set is domain specific and derived from the *DRIOC*, and the solution is a VHDL specification fragment annotated with VSPEC. satisfying the problem description. The case-base is a set of cases and associated indexes used to retrieve cases efficiently.

3.3 Retrieval and Similarity

When presented with a new problem, the case-based synthesis process begins its problem solving activity by retrieving one or more similar cases from the case-base. Retrieval is a three step process of: (a) generating a feature set for the new problem; (b) retrieving functionally correct solutions; and (c) determining the most similar functionally correct solution.

3.3.1 Features and Feature Types

A feature type represents information common to features representing the same characteristic. A set of feature types exists for each case-base. Each feature type has a unique name and describes how to compare features of that type, the relative importance of the feature, and how to generate the feature from a problem description.

The following is the feature type definition for the `input-types` feature. The comparison function is `bag-equal`, its relative weight is 1.0, and `generate-input-types` constructs features of this type from problems.

```
<'input-types, 'bag-equal, 1.0, 'generate-input-types>
```

Sets of features describe problems and facilitate retrieval and comparison of problems. A feature is an attribute value pair where the attribute names a unique feature type and the value is the feature value. A feature is legal if and only if it names a known type. An example of a legal `input-types` feature for an `entity` accepting an integer and a sequence of integers as inputs would be:

```
<'input-types,[''integer'',''seq(element)'']>
```

Features and feature types are defined based on the VSPEC descriptions. A VSPEC description is a collection of logical expressions and argument list definitions when converted. The goal of

the synthesis problem is finding a component whose behaviof and performance meet the VSPEC requirements. In case-based reasoning terms, this translates to finding a component whose VSPEC description is similar to problem requirements and adapting that solution to the specific problem. Because VSPEC is formal, the *DRIOC* elements could be used as features and logical implication used as a matching function - when corresponding elements of two *DRIOC* descriptions are logically equivalent, they match.

The logical equivalence approach to comparing VSPEC descriptions is appealing because feature generation is trivial, the features are general to any domain using VSPEC descriptions, and the comparison is formal. However, logical inference is computationally impractical when considering large case-bases.

The solution is defining features for the specific domain of application, generating those features from VSPEC and using these features for comparison purposes. Generality and formal comparison are lost with this method. However, the efficiency gain from using simple comparisons makes this system far more pragmatic.

A collection of feature types for the DSP domain is currently under development for this system. Following is a short list of some feature types used in further examples:

`input-types`	sequence of input types	`output-types`	sequence of output types
`heat`	heat dissipation	`power`	power consumption
`fft`	Computes FFT	`ordered(x)`	`x` is ordered
`permute(x,y)`	`x` is a permutation of `y`	`search`	is a search system

3.3.2 Feature Generation

When a new problem is presented to the synthesis system, a set of features is generated. The feature generation function from each feature type is applied to the new problem and the resulting features comprise the problem's feature set. Feature generation functions are represented as REFINE functions. The set of generation functions are maintained in a list and applied to each new problem

in a predetermined order to avoid the need for conflict resolution.

The following is the function that generates the `input-types` feature. It returns an attribute value pair consisting of the `input-types` feature name and the value stored in the domain slot of the problem description.

```
function generate-input-types (p : problem) : feature =
  <'input-types,p.domain>;
```

Following is a subset of features generated for the search problem.

```
{<input-types, [sequence(element),integer]>,
 <output-types, [element]>,
 <power, < <=,10>>,
 <search, true>,
 <fft, false>,
 <ordered, false>,
 ...}
```

The first two features represent D and R and indicate what the retrieved case must input and output. The comparison function for each is bag equality indicating the arity and input and output types must match.

Other features are defined based on I and O. No features are generated from I because it is always true. Effectively, there are no input preconditions and the component should work on all inputs of the correct type. The output postcondition, O, does provide information about the desired results of applying this component by defining a search routine.

Finally, features are generated from C. The `constrained by` clause must be a conjunction of simple relations. Each of these relations forms a feature. The type of each feature names the constraint and the relation and value form a pair specifying the value. Simple interval arithmetic is used to compare specific feature values.

3.3.3 Problem Similarity

In a case-based reasoning system, problem similarity indicates the level of confidence that two problems share a solution. This similarity measure is based on two premises. First, the similarity is proportional to the number of common characteristics with matching values. Second, the premise that similarity is proportional to the amount of information involved in the comparison.

The similarity measure implements these premises as *raw similarity* and the *possible match ratio* respectively. Raw similarity is a measure of how many shared features match. Two features *match* if they are of the same type and their values are equal based on the feature type's comparison function. The possible match ratio is a measure of how many feature types are shared by the two feature sets. A feature or feature type is *shared* by two features sets if there is a feature of that type in both sets. Given two sets of features, similarity is the product of the raw similarity value and the possible match ratio.

Raw Similarity

Given two feature sets, raw similarity is the ratio of the sum of weights from matching features to the sum of weights from shared features. Qualitatively, raw similarity determines the weighted percentage of features that match. Formally, raw similarity is defined as the sum weights of matching features divided by the sum of the weights of all shared feature types.

A `DontCare` value in a feature's value slot represents a situation when a feature is present, but its exact value does not matter. More specifically, when the feature contains no useful information for determining problem similarity or is not known. The `DontCare` feature allows the system to distinguish between situations where a feature does not match and when a feature match determination cannot be made.

When a comparison between features of the same type involves a `DontCare` value, a match

always occurs. However, to indicate the inexact nature of the match, the weight used to determine similarity is decreased. In this system, the match is degraded by multiplying the weight from the feature type description by 0.95. Thus, the weight used in the sum of comparison results for raw similarity is 0.95 of its original value. The weight used for calculating the total possible weight is the original weight value.

An example of how the `DontCare` values are used arises when retrieving objects where constraints are not specified. It may be that specific values for a particular constraint are not known because constituent components are not yet described in enough detail. Given a choice between such a component and a component where the constraint is know to be violated, the potential solution should be preferred. By using the `DontCare` feature value instead of a mismatch value or leaving the feature out, the possible solution is preferred over the solution known to be incorrect. If a solution were known to be correct, it would be preferred over the potential solution.

The Possible Match Ratio

The second component of the similarity value, the possible match ratio, is the ratio of the number of shared features to the total number of features defined for the case being considered. The objective of the possible match ratio is to implement specificity in the similarity metric. Given that two cases have equivalent raw similarity values with respect to the current problem, the possible match ratio will prefer the case matching the highest percentage of features, thus involving more information in the comparison.

In addition to preferring more information, the possible match ratio allows loose definition of solution categories. Consider two problems, one described by features specifying input preconditions and output post conditions, and the other described by features specifying representation characteristics. The first feature set describes a problem best solved using a data transform while

the second a problem best solved using a data type. It is conceivable that these two feature sets could share a small number of feature types. If those features match, the raw similarity metric has no means of determining that most features cannot be compared and would return a deceptively high similarity value. The possible match ratio differentiates between these two solution categories because few feature types are shared by the feature sets.

Similarity

The final similarity value is the product of the raw similarity value and the possible match ratio. Table 1 shows the results of comparing two sets of features with the problem's feature set. Note that the second set match is lower due to mismatch of a power consumption feature.

3.3.4 Functional Similarity

The simplest approach to retrieval is calculating a similarity value for each element of the case-based with respect to the problem and choosing the most similar case. The result is a table much like Table 1 extended for the entire case-base. This is not a practical approach for large case-bases due to the complexity of similarity calculation. Thus, solutions matching critical features are retrieved and then ordered using the complete similarity metric.

To accomplish this, the retrieval system maintains indexes for features representing functional characteristics. These features are referred to as *important features.* Following generation of of the problem feature set, important features are extracted. Indexes statically maintained by the retrieval system are used to retrieve a set of cases whose important features match problem features exactly.

Static indexes are created when a case is added to the case-base. Feature values are used as retrieval keys and cases with features that match a particular value can be retrieved directly without a similarity calculation. Important features include `input-types`, `output-types` and some other

features computed from functional requirements. In general, features computed from constraints are not important features, but serve to choose a best solution from all those satisfying the functional requirements.

All features are used to determine final similarity between the initially retrieved set and the problem. Because all potential cases match with respect to important features, features representing other constraints determine the functionally similar case representing is the best solution. The case returned by the retrieval process is the case from the initially retrieved set whose similarity with the problem is maximal.

The two stage retrieval process results from two observations. First, the belief that design is a process of finding a set of functionally correct solutions, then using problem constraints to select from them an optimal solution. Important features indicate the primary function of the artifact. The remaining features describe the constraints the solution exists under. Second, initial retrieval is achieved using statically maintained indexes, without the cost of calculating similarity. Similarity is calculated over this subset of the original case-base. This dramatically reduces retrieval cost with respect to a brute force approach that calculates similarity for every element in the case-base.

Consider the feature sets generated for linear search and batch sequential search shown in Figure 5. Using only important features, these two cases are identical. They both search arrays of elements and return the indicated element if found. Thus, the initial retrieval would return both, but eliminate cases for sorting, FFTs, and cases where range and domain are not matches.

Although the two solutions are functionally equivalent, the `power` feature representing a constraint differs. In the linear search `entity` the power constraint from the original specification is violated while in the batch sequential `entity` the power is not known. (See Table 1 for exact similarity calculations.) Thus, the similarity of the linear search case is lessened and the batch sequential option is preferred. The power constraint is not verifiable, but unlike the linear search

option, it is not known that the constraint is violated. This is an example of using the DontCare feature value to indicate a situation between a perfect match and a mismatch.

The batch sequential search architecture returned is the same as the architecture shown in Figure 3. Note that a behavioral specification of the bin_search entity exists, however no specification for the sort entity exists aside from the VSPEC description. This represents a complete specification of the problem and can be viewed as a solution. However, it is possible to repeat the process and attempt to synthesize a specific component for the sort description. This is achieved during adaptation by repeating the synthesis process using requirements from the sort description.

3.4 Adaptation

The most similar case found by the retrieval process is modified to fit the current problem by the adaptation process. Adaptation employs two independent methods. The first is application of adaptation rules. The second is replacement of case parts by generating a sub-problem and recursively calling the case-based reasoner.

3.4.1 Rule-Based Adaptation

An adaptation rule is a REFINE transform. The antecedent is a predicate accepting three arguments: the problem being solved, the problem solved by the case, and the specification fragment being adapted. The consequent is a REFINE predicate accepting the same arguments that implements the change to the specification fragment. A list of adaptation rules is maintained by the system. Each rule is evaluated during the first stage of adaptation. Conflict resolution is achieved using a static ordering based entirely on the order rules exist in the rule-base.

VHDL and VSPEC components are stored using an object-based abstract syntax tree common to REFINE parsing activities [1]. This representation makes application of adaptation rules much

easier because the object model is manipulated rather than plain text. The advantage is that adaptation rules need not parse text to perform their operations. Retrieving the source code from the object model simply requires calling a single print routine, thus there is no loss of solution generality.

An example of a frequently used adaptation rule does variable substitution. This function gathers all identifiers and references from an `entity` structure and applies a transformation to them. In a semantically correct abstract syntax tree, each variable has a definition and several references. To change the name of a variable, the name must be changed at the definition point and each reference point. This transformation is called on each node in the abstract syntax tree. If a node is an identifier reference, it checks the identifier name and changes those matching the old variable name to the new variable name. Similarly, it finds the identifier definition and changes its name to the new name. Without the abstract syntax tree, the source VSPEC would require lexical and syntactic analysis to perform this operation.

```
function subst-variable (the-entity:entity,old-name:symbol,new-name:symbol) =
  let (idents = entity-port(the-entity),
       idents-refs = descendants-of-class(the-entity,'ident-ref))
    ref in idents-refs &
    ident-name(ref) = old-name &
   -->
    ident-name(ref) = new-name;
    ref in idents &
    ident-name(ref) = old-name
   -->
    ident-name(ref) = new-name
```

3.4.2 Sub-Problem Based Adaptation

The second adaptation method is case element replacement. This involves defining a function or goal associated with the fragment and using the internal environment defined by the case to constrain possible solutions [6].

Case fragment replacement is used to replace a portion of a structural specification `architecture`. Because a structural `architecture` is a collection of components, identification of a case fragment for replacement is identifying an appropriate component. To define a function for the case fragment, the reasoning process uses the difference between the system state before and after the execution of the component action. The reasoning process assumes that the component caused the difference intentionally. Thus, the difference defines what must be the goal of the component. Constraints defined by preconditions and the external environment together defined constraints on the new synthesis problem. The difference between the system state before and after component execution is obtained from either the VSPEC representation of the component, or from VSPEC defining outputs of systems providing input to the component and the preconditions of components receiving output. The result is a new problem whose solution can replace the original.[3]

Replacing components also may occur when instantiating a general architecture. Recall that `entity` structures referenced by an `architecture` may be defined only using VSPEC and need not have a VHDL implementation. Thus, the requirements of the component are expressed without the specifics of the implementation. The VSPEC is transformed into a problem description and a VHDL component satisfying the requirements results.

As an example of case component replacement, consider synthesis of an `architecture` for the `sort entity`. The *DRIOC* form of the VSPEC is as follows:

$D =$ $\quad seqence(element) \times integer$

$R =$ $\quad element$

$I(x) =$ $\quad true$

$O(x, z) =$ $\quad \forall e : element \bullet output = e \Leftrightarrow e \in input \wedge k = key(e)$

$C =$ $\quad power \leq 10$

[3]For a detailed discussion of this adaptation scheme, please see [4]

The feature set associated with this problem is similar to the feature set for the original search problem, but no constraints are specified and feature values are changed appropriately.

```
{<input-types, [sequence(element)]>,
 <output-types, [sequence(element)]>,
 <power, DontCare>,
 <search, false>,
 <fft, false>,
 <ordered, true>,
 <permutation, true>,
 ...}
```

The retrieval activity here is identical to retrieval of the batch sequential architecture and the discussion will not be repeated. Any appropriate sorting `architecture` may be retrieved given the current set of features. For this example, assume a `quicksort entity` is retrieved described by the VSPEC `entity` shown in Figure 6.

The resulting VHDL description is the batch sequential `architecture` combined with the `quicksort architecture`This represents a new solution at a lower level of abstraction. Before it is accepted as a solution, the new system must be evaluated with respect to constraints.

3.5 Evaluation

Following synthesis of the potential solution, a case-based reasoner attempts to evaluate a solution to determine its fitness. The evaluation process in this case-based reasoning system exclusively involves determining if the proposed solution meets specified constraints.

Recall that the `constrained by` clause defines constraints the system must satisfy. These constraints are translated into features for the retrieval process. If the solution is monolithic, constraint satisfaction is a simple comparison of $C(c_1 \ldots c_n)$ from the problem description and the proposed solution. However, when the solution is a collection of components, more complex constraint verification must be applied.

Constraint verification is achieved by specifying constraint behavior and transforming that behavioral description into REFINE theories. Given the constraints from the high level specification and a set of constraints from component constraints, the REFINE theories determine if the composition of component constraints continue to meet the higher level constraints. Theories currently exist in this system to evaluate heat dissipation, power consumption, clock speed, pin-to-pin timing, and area. By checking performance constraints in the earliest stages of synthesis, such issues are managed concurrently with the synthesis activities.

With the batch sequential search system completed, constraints on the subcomponents of the batch sequential search algorithm are now known. The theory of power consumption this system uses states that the total power used by a device is equal to the sum of the power used by the device's components. Instantiated for this problem, the total power used by `bin_search` and `quicksort` must be less than 10 watts. The constraints on the components say that they consume no more than 4 watts and no more than 5 watts respectively. Using interval arithmetic, the sum is no more than 9 watts and the 10 watt constraint is met. If a constraint violation is discovered, the offending potential solution is discarded. The reasoning process is repeated in an attempt to find a better solution. Alternative approaches include simply reporting the constraint violation or involving the user in the decision process.

Evaluation of constraints occurs both when retrieving and evaluating potential solutions. At each stage of the synthesis activity, non-functional requirements are evaluated concurrently with functional requirements. Thus implementing a simple concurrent engineering synthesis process.

4 Limitations

Early experimentation indicates this synthesis approach is effective using small case-bases in reasonably restricted domains. Currently, this approach is being extended to solve co-design problems and the initial case-based is being extended. Several limitations, although not fatal, have been identified.

4.1 Case-Base Construction

The greatest potential limitation to this approach is case-base construction. The system cannot use a component that is not defined in its case-base, implying that a large case-base must be developed, or individual case-bases must be developed for each domain. This requires identification of a core set of components with VSPEC annotations.

VHDL libraries currently exist and are being aggressively constructed in the DSP domain. However, these libraries are not annotated with VSPEC, thus forcing back annotation by hand or using an automated approach. Hand annotation is time consuming and difficult. Automated annotation is not practical at this time.

An alternative approach is implementation of other synthesis techniques that generate innovative solutions and use these approaches to augment the case-base. Currently this approach is being pursued through integration with the KIDS software synthesis environment and transformation based synthesis techniques. New solutions are generated when necessary and old solutions are re-used when possible. It should be noted that although they do extend the case-base, other techniques are also limited by their synthesis knowledge.

4.2 Features and Feature Generation

Each synthesis domain requires definition of feature types and feature generation functions. Once cases are identified, they must be indexed and stored in the case-base. As with case-based construction, a universal set of features can be defined, or individual feature sets can be developed for each domain. The second solution is the obvious choice given the trade-off between computational complexity and brittleness caused by domain specific features. Without exploiting some domain specific information, retrieval become computationally prohibitive.

4.3 Solution Correctness

Currently there is no guarantee that any given solution is correct. If a VHDL solution is synthesized, simulation is available for some limited correctness evaluation. An approach currently being developed is to use a theorem proving approach on the VSPEC description. The limitations of such an approach as a retrieval technique were presented earlier. However, once a solution is found, the problem is reduced to checking one solution. This still requires pragmatic, efficient theorem proving techniques to ultimately be practical.

5 Related Work

As VSPEC is a Larch interface language for VHDL it borrows from the construction of other interface languages. Specifically, VSPEC is styled after the LM3 Larch interface language for Modula-3 [10]. Odyssey Research Associates is currently developing an alternative Larch interface language for VHDL [9]. This language does not support representation of constraints other than time and is targeted for formal analysis rather than synthesis. They are attempting to generate a formal semantics for VHDL using LSL for proving correctness. ORA's interface language also differs in its

implementation of time. An absolute time based temporal logic is used in specifying the function of an `entity`. Thus one can specify that a predicate becomes true at a specific time using the notation "$P(x)@t$". The VSPEC notation specifies time intervals as constraints independent of system function.

Another attempt to annotate VHDL is VAL [5]. VAL annotates all aspects of the VHDL design. All signals in the namespace of the VHDL representation are in the namespace of the VAL annotation. Thus, VAL annotates specific VHDL designs rather than represent requirements. ORA's interface language is similar in this respect, but does support separate requirements definitions.

Although VHDL is a hardware synthesis language, synthesis of VHDL designs is a software synthesis activity. Viewing software synthesis as a planning activity was proposed in the KBSA effort [16, 22, 17] and used heavily in the BENTON [2] system. Both systems use plans to represent and control software synthesis activities. In this system, plans are not explicitly used and represent only the structure of solutions. Other attempts at case-based software design include CEASAR [7], analogical reuse [11, 12], and work in derivational analogy [13]. Some also view reuse work by Prieto-Diaz [15] as case-based reasoning, however it is not a heuristic approach and involves no adaptation of final solutions.

6 Future Work

Current VSPEC research involves pursuing domain specific support for specification activities and support for formal synthesis. An important aspect of any Larch language is its associated handbook. A handbook is simply a collection of reusable theories defined in the shared language. Handbook theories represent commonly used structures, algorithms and characteristics as well as domain specific information. For VHDL theories to represent standard VHDL types, low level logic functions and

conversion routines are being implemented. In addition, libraries to support specifications involving signal attributes such as `event`, `stable`, and `delay` are under development. Theories for pin-to-pin timing, heat dissipation, power consumption, area and clock speed have been implemented to support constraint checking during the design process.

The isomorphic relationship between VSPEC and algebraic specifications is being used to exploit work in formal synthesis, specifically, developing morphisms between algorithms [21]. This involves development and implementation of theories useful in constructing multicomponent systems such as the batch sequential search algorithm appearing earlier in this paper.

Finally, formal evaluation of specifications and solutions is being explored. Although it may be impractical to use formal inference in the retrieval process, once a solution is found it is a practical verification tool. Given VSPEC descriptions of both the problem and solution, various levels of satisfaction may be evaluated. Logical equivalence is the ideal comparison, however, logical implication may be acceptable. Even more interesting is the use of modal logics and antecedent discovery to correct incomplete specifications or restrict solutions.

7 Acknowledgments

Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615-93-C-1316. The authors wish to thank Wright Labs and ARPA for their continuing support and direction. The authors also wish to thank Hal Carter, Philip Wilsey, Ranga Vemuri and Paul Bailor for their invaluable comments on (and criticism of) the VSPEC language. A final word of thanks to the reviewers for their comments and suggestions.

References

[1] L. Abraido-Fandino. An overview of refine 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering*, Madrid, Spain, April 1987.

[2] P. Alexander. BENTON: A Multi-Agent System for Larch Specification Generation. In *The 5th International Conference on Software Engineering and Knowledge Engineering*, pages 125–133, San Francisco, CA, June 1993. Knowledge Systems Institute.

[3] P. Alexander. Combining transformational and derivational analogy in Larch specification generation. In *Proceedings of The 6th International Conference on Software Engineering and Knowledge Engineering*, pages 131–138, Riga, Latvia, June 1994. Knowledge Systems Institute.

[4] P. Alexander and C. Tsatsoulis. ASP-II: An Experiment in Combining Case-Based and Skeletal Planning. *International Journal of Expert Systems: Research and Applications*, 4(2):221–247, 1991.

[5] L. Augustin, D. Luckham, B. Gennart, Y. Huh, and A. Stanculescu. *Hardware Design and Simulation in VAL/VHDL.* Kluwer Academic Publishers, Boston, MA, 1991.

[6] Ralph Barletta and William Mark. Breaking Cases Into Pieces. In *AAAI Case-Based Reasoning Workshop*, pages 12–16, Minneapolis-St. Paul, 1988. AAAI.

[7] G. Fouqué and S. Matwin. CEASAR: a system for CAse basEd SoftwAre Reuse. In *Proceedings: 7th Annual Knowledge-Based Software Engineering Conference*, pages 90–99, McLean, VA, September 1992. IEEE Computer Society Press.

[8] J. Guttag and J. Horning. *Larch: Languages and tools for formal specification.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1993.

[9] D. Jamsek and M. Bickford. Formal Verification of VHDL Models. Technical Report RL-TR-94-3, Rome Laboratory, Griffiss Air Force Base, NY, March 1994.

[10] K. Jones. LM3: A Larch Interface Language for Modula-3. Technical Report 72, DEC Systems Research Center, Palo Alto, CA, 1991.

[11] N. Maiden and A. Sutcliffe. Analogical Matching for Specification Reuse. In *Proceedings: 6th Annual Knowledge-Based Software Engineering Conference*, pages 101–112, Griffiss AFB, NY, September 1991. IEEE Computer Society Press.

[12] S. Meggendorfer and P. Manhart. A Knowledge and Deduction Based Software Retrieval Tool. In *6th Annual Knowledge-Based Software Engineering Conference*, pages 126–137. IEEE Computer Society Press, 1991.

[13] K. Miriyala and M. Harandi. The Role of Analogy in Specification Derivation. In *Proceedings: 6th Annual Knowledge-Based Software Engineering Conference*, pages 113–125, Griffiss AFB, NY, September 1991. IEEE Computer Society Press.

[14] D. Perry. *VHDL*. McGraw-Hill, New York, NY, 1st edition, 1991.

[15] R. Prieto-Diaz. Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 34(5):88–97, 1991.

[16] C. Rich. A Formal Representation for Plans in the Programmer's Apprentice. In *7th International Joint Conference on Artificial Intelligence*. Morgan Kaufman, 1981.

[17] C. Rich and Y. A. Feldman. Seven Layers of Knowledge Representation and Reasoning in Support of Software Development. *IEEE Transactions on Software Engineering*, 18(6):451–469, 1992.

[18] D. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, Sept. 1985.

[19] D. Smith. Algorithm Theories and Design Tactics. *Science of Computer Programming*, 14:305–321, 1990.

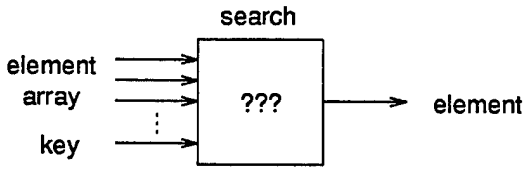
[20] D. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, Sept. 1990.

[21] D. Smith. Classification approach to design. Technical Report KES.U.93.4, Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA, November 1993.

[22] R. Waters. The Programmer's Apprentice: A Session with KBEmacs. *IEEE Transactions on Software Engineering*, 11(11):1,296–1,320, 1985.

```
entity search is
  port (input: in array of element;
        k: in keytype;
        output: out element);
end search;
```

a)

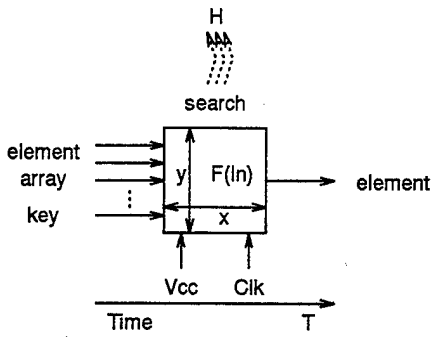


b)

Figure 1: A VHDL entity describing a record search.

```
entity search is
  port (input: in array of element;
        k: in keytype;
        output: out element);
  modifies output;
  requires true;
  ensures
    output = e <=> key(e)=k and
                   e in input
  constrained by
    power =< 5 and
    area =< .3
end search;
```

a)



b)

Figure 2: A VSPEC entity describing a record search.

Feature Name	*Problem*	*Linear Search*	*Batch Sequential*
goal	entity	entity	entity
input-types	[integer,seq(element)]	[integer,seq(element)]	[integer,seq(element)]
output-types	[element]	[element]	[element]
fft	false	false	false
ordered(z)	DontCare	DontCare	DontCare
permute(x,z)	DontCare	DontCare	DontCare
search	true	true	true
heat	DontCare	DontCare	DontCare
power	<<=,10>	DontCare	<<=,11>
area	DontCare	DontCare	DontCare
Possible Match	1.0	1.0	1.0
Raw Similarity	1.0	0.978	0.975
Similarity	1.0	0.978	0.975

Table 1: Table showing a subset of features generated for the search problem and 2 potential solutions. The bottom rows indicate calculated similarity values. Assume all weights are 1.

```
entity search is
  port (input: in array of element;
        k: in keytype;
        output: out element);
  modifies output;
  requires true;
  ensures
    output = e <=> key(e)=k and
                   e in input
  constrained by
    power =< 5 and
    area =< .3
end search;

architecture structure of search is
  component sorter
    port (input: in array of element;
          output: out array of element);
  component bin_search
    port (input: in array of element;
          key: in keytype;
          value: out element);
  signal sorted_array: array of element;
begin
  sort_instant: sorter
    port map (input=>in_array;
              output=>sorted_array);
  search_instant: bin_search
    port map (input=>sorted_array;
              k=>in_key;
              value=>out_value);
end bat-seq;

entity sort is
  port (input: in array of element;
        output: out array of element);
  modifies output;
  ensures bag(input) = bag(output) and
          sorted(output);
  constrained by
    power <= 2 and
    area <= .1;
end sort;
```

```
entity bin_search is
  port (input: buffer array of element;
        k: in keytype;
        value: out element);
  modifies out;
  requires sorted(input);
  ensures
    (fa e:element)
      output = e <=> key(e)=k and
                     e in input;
  constrained by
    power <= 3 and
    area <= .2;
end bin_search;

architecture behavior of bin_search is
  begin
    p1: process
          -- VHDL representation of a
          -- binary search over ordered
          -- arrays
    end process;
end behavior;
```

Figure 3: VSPEC representation of a search architecture using a batch sequential approach. The original list is sorted and a binary search finds the desired object from the resulting list.

```
entity search is
  port (list: in array of element;
        k: in integer;
        output: out element);
  modifies output;
  requires true;
  ensures
    (fa e:element)
      (output = e) <=>
        (e in input and
         k = key(e));
  constrained by
    power <= 10;
end example;
```

Figure 4: VSPEC requirements for a searching component.

```
{<input-types, [sequence(element),integer]>,
 <output-types, [element]>,
 <power, DontCare>,
 <search, true>,
 <fft, false>,
 <ordered, false>,
 ...}
```

a) Features from batch sequential search

```
{<input-types, [sequence(element),integer]>,
 <output-types, [element]>,
 <power, <=,11>>,
 <search, true>,
 <fft, false>,
 <ordered, false>,
 ...}
```

b) Features from linear search

Figure 5: Features from linear and batch sequential search returned by the retrieval algorithm.

```
entity quicksort is
  port (input : in array of element;
        output : out array of element)
  modifies output;
  requires true;
  ensures
    bag(input)=bag(output) and
    sorted(output);
  constrained by
    power <= 5;
end quicksort;
```

```
architecture behavior of quicksort is
  begin
    p1: process
         -- VHDL description of a
         -- quicksort algorithm
    end process;
end behavior;
```

Figure 6: quicksort entity with VSPEC annotations and behavioral VHDL description.

APPENDIX M:

Extending VHDL to the Systems Level*

Perry Alexander and Phillip Baraona
Department of Electrical and Computer Engineering
and Computer Science
PO Box 210030
The University of Cincinnati
Cincinnati, OH 45221-0030
{alex,pbaraona}@ececs.uc.edu
www.ececs.uc.edu/~kbse

Abstract

Systems engineering is the process of looking at many facets of an emerging design. A systems engineer is required to examine and reconcile many information sources when making high level design decisions. Although VHDL *is an excellent digital system description language, it lacks flexibility to address all systems level issues. Digital system behavior and structure are effectively handled, but other facets are not.* VSPEC *represents one attempt to model other facets in the* VHDL *framework. It adds functional requirement and performance constraint modeling to the* VHDL*-based design process. This paper first describes* VSPEC *and its interaction with* VHDL*. It argues that* VSPEC *is an excellent first step towards a systems level description language. However, other facets are needed to model complete systems. A language structure for representing these facets is proposed and a potential source for a semantic definition is identified.*

1 Introduction

Systems level design is characterized by the need to deal with heterogeneity during the design process. Heterogeneity arises from two sources: (i) modeling components using different computational models; and (ii) modeling different component and system facets. Different system components are best modeled using different basic semantic models. Digital electronic, analog electronic, optical, and MEMS components all have different underlying mathematical domain models. Like heterogeneous components, multiple facets of the same component require different underlying semantic models. Electromagnetic, analog, digital and constraint facets again have different underlying mathematical domain models. Further, different models may be used for the same facet under different circumstances.

To address the systems level design process, VHDL must be extended to include: (i) multiple modeling paradigms for different component facets; (ii) multiple modeling paradigms for different component domains; and (iii) a means for moving information between system representations. Multiple modeling paradigms supports integrated modeling. Moving information between system representations supports using multiple semantic models without forcing a single model. Interestingly, VHDL provides syntactic support for multiple modeling paradigms. The `entity/architecture` model supports defining both structural and behavioral models for the same component. This basic architecture has been used to extend VHDL to the analog domain and to define constraint and requirements models.

Moving information between semantic models presents a more difficult problem. Effectively, the VHDL semantics must be extended. Goguen's model of institutions [6] can be used as a basis for such modeling. Using institutions, semantic domains are defined as categories and functors used to define when information from one domain is valid in another.

This paper presents existing efforts to move VHDL to the systems level. First, a brief overview of VSPEC is presented. VSPEC is an interface specification language for VHDL that represents an initial attempt to model multiple component facets at the requirements level. Second, the model associating an `entity` with one or more `architecture` is extended to provide a multi-faceted model. As an example, the VSPEC requirements and constraint models are repre-

*Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615-93-C-1316

sented. The paper concludes by describing open semantic issues and problems that must be addressed.

2 VSPEC - A First Step

VHDL provides users with a means for modeling both the behavior and structure of a digital system. It provides users with an operational language for describing the behavior of a component. This language subset, referred to as *behavioral* VHDL, allows users to describe data transforms and control structures for components using a programming language style syntax. VHDL also provides users with a declarative language for describing the structure of a system. This language, referred to as *structural* VHDL, allows users to describe interconnections between components using a simple net list-based module interconnect language.

Using behavioral and structural `architectures` of the same component allows VHDL users to model two facets of components and systems: function and structure. Thus, a user might provide a high level, black-box behavioral description and use that description as a basis for refinement into a structural system decomposition. Such activities are common in top-down design processes making these facets and their interaction quite useful to systems designers.

Behavioral and structure VHDL share a common simulation-based semantics that allows information from one facet to be visible in the other. More specifically, the results of simulating structural and behavioral representations of a component can be directly compared. Thus, designers are able to evaluate the results of design iterations by simulating and comparing results.

Although VHDL has excellent operational specification capabilities, their application during the design process is limited. One limitation noted in our research activities is at the abstract requirements specification level. Specifically: (i) VHDL's operational semantics are not suitable for abstract functional requirements; and (ii) VHDL provides no means for describing performance requirements. These two information classes form important information facets useful in the design process. VSPEC is an initial attempt to address these facets in the context of VHDL.

2.1 An Example

VSPEC uses a modified axiomatic specification technique for modeling a component's function and performance requirements. A pre- and post-condition are defined to indicate: (i) what must be true in the current state; and (ii) what must be true in the next state. This pre- and post-condition follow the traditional axiomatic semantics presented by Hoare [9] and are implemented using a Larch Interface Language approach [8]. This axiomatic specification is augmented with an activation condition indicating what circumstances cause the component to activate. The activation condition is needed because of the concurrent nature of VHDL components in contrast to the serial nature of software components.

The axiomatic specification approach is further modified to describe performance requirements. Such performance requirements are modeled using a simple declarative semantics to express relations over constraint variables. They are effectively invariants with respect to the axiomatic functional requirements.

To understand how VSPEC defines requirements and constraints, an example of a simple search component is presented in Figure 1. This component accepts an array of elements and a key and returns the array element associated with that key. Changing the value of either the key input or the array to be searched causes the component to activate.

```
package search_types is
  type E is mutable;
  type K is mutable;
  type E_array is array (integer range <>) of E;
end search_types;

use work.search_types;
entity search is port
  (input: in E_array;
   k: in K;
   output: out E);
  includes KeyToElement(E, K);
  includes Area, Power, Frequency;
  modifies output;
  sensitive to
    k'event or input'event;
  requires true;
  ensures forall e: E
    ((output'post = e) iff
       (key(e)=k
       and e ∈ input));
  constrained by
    area ≤ (3 um * 5 um)
    and power ≤ 10 mW
    and clock_frequency ≤ 50 MHz;
end search;

KeyToElement(E,K): trait
introduces
  key: E → K
```

Figure 1. An example component defining requirements for a simple search component.

2.2 Functional Requirements

The basic specification model used for VSPEC functional requirements is a state machine. Figure 3 represents in-

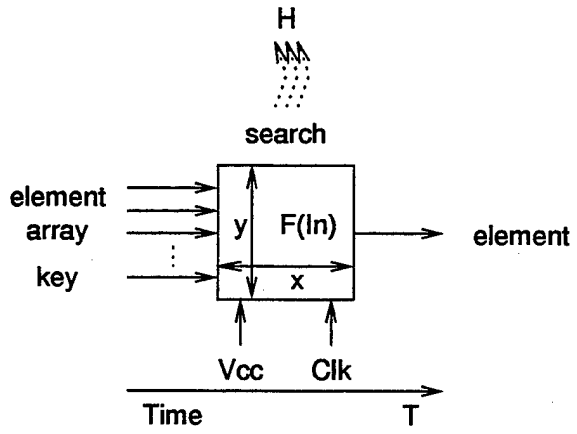


Figure 2. A graphical representation of VSPEC information representation.

formation defined by a VSPEC component. Using the axiomatic style, a state machine is specified. Pre-conditions and post-conditions define the output and next state functions while entity ports and VSPEC state variables define component state.

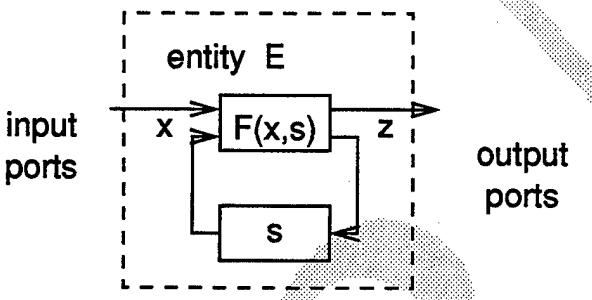


Figure 3. State-based model of functional specification.

Functional requirements are modeled using the activation condition, pre-condition and post-condition. These are specified in the `sensitive to`, `requires`, and `ensures` clauses, respectively. The `requires` clause defines a pre-condition that must be true in the current state for the component to execute correctly. The `ensures` clause defines a post-condition the component must make true in the next state given the pre-condition is true in the current state. Given that $\vec{x}$ is the collection of `entity` ports and VSPEC state variables providing input or state and $\vec{z}$ is the collection of `entity` ports and VSPEC state variables providing output or next state, the relationship defined by the `requires` and `ensures` clauses can be defined as:

$$\forall \vec{x} \cdot \texttt{requires}(\vec{x}) \Rightarrow \exists \vec{z} \cdot \texttt{ensures}(\vec{x}, \vec{z}) \quad (1)$$

The axiomatic specifications define the data transformation performed by the component. These specifications define requirements on how the component transforms data by defining relations between inputs, current state and outputs. Specifically, any implementation of these requirements, $F(\vec{x})$, must provide a witness for $\vec{z}$ that satisifies the requirements. Skolemizing Equation 1 results in the following correctness condition for the data transformation:

$$\forall \vec{x} \cdot \texttt{requires}(\vec{x}) \Rightarrow \texttt{ensures}(\vec{x}, F(\vec{z})) \quad (2)$$

Although simple, the importance of this relationship is the connection it provides between the requirements defined by VSPEC and the execution of a VHDL implementation. Given only these requirements, any VHDL implementation of $F(\vec{x})$ is a correct implementation. Thus, the requirements facet is connected semantically to the behavioral or structural facet. Further, the requirements facet could be associated with a test facet or other facet defined for a component.

The `ensures` and `requires` clauses of the example `search` component (Figure 1) define the following axiomatic requirements:

$$\forall input : E_array, k : K \cdot true \Rightarrow \forall e : E, \cdot \exists output : E \cdot \quad (3)$$
$$output = e \Leftrightarrow key(e) = k \wedge e \in input$$

Simplifying the implication via implication elmination yields:

$$\forall input : E_array, k : K, e : E, \cdot \exists output : E \cdot \quad (4)$$
$$output = e \Leftrightarrow key(e) = k \wedge e \in input$$

The requirement defined in Equation 4 states that an output of this component is correct if and only if: (i) the output is in the input set; and (ii) the key associated with the output is equal to the input key. Any component meeting this requirement is potentially a solution to the defined problem. Note that using the declarative representation, requirements are stated directly rather than identifying a specific candidate solution.

The activation condition defined in the `sensitive to` clause defines when a component becomes active. Like the pre-condition, the activation condition must be true for the component to function. If the pre-condition is false, the component's behavior is undefined. In contrast, if the activiation condition is false the component maintains its current state. The activiation condition models events that cause the component to perform its task.

When components are interconnected, activation conditions model interaction between components. Activation conditions are defined over the same symbol set as preconditions. They monitor inputs and state to determine when the component should perform its data transform. When inputs are connected to outputs from other components or inputs from outside the system, control is communicated between components. Activation conditions are modeled using a process algebra. Process algebras are designed specifically to model reactive systems and suit the semantic needs of activation conditions nicely. Specifically, VSPEC activatcion conditions are modeled using CSP [10].

Each VSPEC entity is modeled as a CSP process. The alphabet of each process' is the set of system states where its associated activation condition is true. By definition, the CSP process ignores any symbol not in its alphabet. Thus, any state where the component is not active is ignored by the component.

Given an activation condition $A(\vec{x})$, the set of states where A is active is defined as $\Psi_A = \{s : S \mid A(s)\}$. Using Ψ_A the process P associated with a VSPEC component is defined loosely as:[1]

$$P_s = e : \Psi_A \rightarrow P_{s'} \quad (5)$$

where s and s' are the current and next states and satisfy the axiomatic requirements. Briefly, the notation indicates that a process, P in state s waits for an event e from Ψ_A. Because Ψ_A only contains states where the activation condition is true, P will effectively ignore all other states. For any e in Ψ_A, a process P in state s' results. If it is known that some function F satisfies the axiomatic requirements, then the previous equation can be rewritten as:

$$P_s = e : \Psi_A \rightarrow P_{F(s)} \quad (6)$$

Note that even within VSPEC's functional modeling component, two facets exist. Specifically, the axiomatic model of data transform and the process algebra model of control. In the semantics of VSPEC, these two facets communicate by sharing a common definition in the Larch Shared Language [8].

The `sesitive to` clause from the `search` component (Figure 1) defines the following activation condition:

$$k'event \vee input'event \quad (7)$$

Removing the syntactic sugar gives the following predicate:

$$\forall k : K, input : E_array \cdot event(k) \vee event(input) \quad (8)$$

[1] Component semantics are substantially more complex than this simple example. The complexity adds nothing to this paper. Interested readers should reference specific VSPEC papers listed in the bibliography [4, 3]

The attribute `event` is actually a predicate that is true when the value of its assocated symbol has changed from the last state. Thus, the activation condition is true when either the key or search database changes values.

2.3 Architectures

An architecture is a collection of interacting components. VHDL provides structural descriptions for defining component and process interconnection. VSPEC uses the same structural descriptions to connect entities annotated with VSPEC definitions. A VSPEC structural description is exactly analogous to structural architectures used in traditional VHDL. Figure 4 shows a VSPEC component architecture for a `search` architecture. This architecture specifies a sort component that prepares input for a binary search component. Figure 5 defines the VSPEC requirements for the components used in the architecture.

```
use work.search_types;
architecture structure of search is
  component sort
    port (list_in: in E_array;
          list_out: out E_array);
  end component;
  component bin_search
    port (list_in: in E_array;
          k: in K;
          e: out E);
  end component;
  signal x: E_array;
begin
  C1: sort port map (input,x);
  C2: bin_search port map (x,k,output);
end structure;
```

Figure 4. A candidate architecture for the search example.

VSPEC's process algebra semantics supports defining bisimulation relationships [14] between single component requirements and VSPEC component architectures. A VSPEC architecture is a decomposition of a system into a collection of interconnected components where the requirements of each component are known but an implementation has not yet been defined. A VSPEC architecture represents a decompsition step in a top down design process. Bisimulation relationships define when a VSPEC architecture exhibits behavioral equivalence with its associated requirements; *e.g.* when they look the same at their interfaces. Thus, using the axiomatic semantics of VSPEC with its process algebra control semantics a structural facet (the `vspec`

architecture) can be related with a requirements facet (the component specification).

```
use work.search_types;
entity sort is port
  (list_in: in array of E;
   list_out: out array of E);
  sensitive to list_in'event;
  requires true;
  ensures
    ordered(list_out'post) and
    perumuation(list_in,list_out'post);
end bin_search;

use work.search_types;
entity bin_search is port
  (list_in: in array of E;
   k: in K; e: out E);
  sensitive to
    list_in'event or k'event;
  requires ordered(list_in);
  ensures ∀ f: E
    output'post = f iff
       key(f)=k
       and f ∈ input;
end bin_search;
```

Figure 5. Component specifications for candidate search architecture.

2.4 Performance Constraints

Performance requirements are modeled using relations defined in the `constrained by` clause. These relations define constraints over a collection of variables defining constraint types. The component is required to meet those constraints at all times in every state. Thus, constraints behave much like invariants with respect to the axiomatic functional requirements.

The semantics of constraints can be defined in terms of a component's state. Simply, the constraint predicate must be true for all states:

$$\forall s : S \cdot C(s) \tag{9}$$

The `constrained by` clause from the `search` example (Figure 1) defines the following constraint predicate:

$$\forall s : S \cdot area <= (3um * 5um) \tag{10}$$

$$\wedge power <= 10mW \tag{11}$$

$$\wedge clock_frequency <= 50MH$$

In VSPEC, physical types behave like VHDL physical types. Thus, these relations define constraints on area, power consumption and clock speed.

Constraints present special problems when interacting with other facets. Requirements, behavior and structural facets interact in relatively intuitive ways. Providing proper semantics defines clean relationships between facets. Constraints do not share this characteristic. Constraint variables (*e.g.* heat and area) have no analog in any other facet. Further, it is difficult if not impossible to model the relationship between a functional requirement and a constraint. Constraints have neither a simulation or true axiomatic semantic making relationships difficult to define.

3 VHDL, VSPEC and Systems Level Design

VDHL provides two facets for modeling digital systems: (i) behavioral; and (ii) structural. The `entity/architecture` pair structure provides means for associating multiple facets to the same interface. However, VHDL provides only a simulation semantics for representing systems. This limits VHDL's impact in multi-facetted modeling at the systems level.

VSPEC adds new facets and new modeling paradigms for those facets. The initial objectives for designing VSPEC were to support very high level synthesis. Specifically, capabilities for specifying: (i) declarative functional requirements; and (ii) performance constraints were initially developed. Activiation conditions and architectures followed as the need to represent component decomposition became apparent.

VSPEC demonstrates the effectiveness of multi-facetted modeling. By adding modeling capabilities that do not require operational semantics, support is provided for modeling requirements declaratively. Because requirements define "what" rather than "how", declarative semantics make sense for requirements modeling. Further, a declarative semantics extended to both performance constraints and function.

Looking back at systems level language requirements and examining VSPEC and VHDL reveals that several systems level modeling requirements are met. Both VHDL and VSPEC provide support for multiple modeling paradigms for different component facets. VHDL provides behavioral and structural support using operational semantics. VSPEC provides fucntional requirements and performance constraint support using a declarative semantics. VHDL and VSPEC also support modeling different components in the same system using different computational models. The semantics for achieving this are still somewhat arcane, but they do exist and are usable. Finally, a limited means for moving information between facets exists. VHDL uses a single, common operational semantics while VSPEC uses a sin-

gle, common declarative semantics. Bisimulation provides a link between VHDL and the functional aspects of VSPEC. Links to and from the constraint aspects of VSPEC are not as well defined.

4 Moving VHDL to Systems Level Design

Moving VHDL to the systems level involves taking the concepts demonstrated in VSPEC and: (i) extending them to the general case; and (ii) providing consistent language support. This section describes one possible method to accomplish these goals. This description represents initial thoughts on this topic: none of the VHDL extensions described in this section have been implemented.

Extending the facet concept to more general cases means providing a general structure for supporting facets. VSPEC currently annotates the `entity` description because it defines interface requirements. Thus, the interface is the most logical place for the descrption. The component interface is not the best place for describing all requirements.

VHDL does provide a structure useful for assigning multiple models to a component. The `entity/architecture` model allows multiple models to be defined for a given interface. To extend this, language support must be provided for different facets of an entity. Specifically, the `architecture` is replaced by a `facet` structure that serves a similar, more general purpose. Figure 6 shows several facets defined for a single entity.

Each facet defined in Figure 6 uses its own computational model. The model is selected based on appropriateness for the information being represented and language constructs are provided appropriately. As new facets are identified, new facets are added to the systems level language using this common syntactic support. The heterogeneous nature of facets makes them substantially different than VHDL architectures all of which share a common simulation based semantics.

5 A Potential Semantic Basis

The need to move information between facets is what defines systems engineering activities. How those heterogeneous models interacts profoundly influences work at the systems level. Thus, it is important to begin modeling the interaction of facets. The syntax described in the previous section that extends VHDL to the systems level is rather standard. However, mixing computational models within the same language environment presents interesting research challenges. Reconciling information between computational models may be the most difficult of these challenges.

The approach taken by both VHDL and VSPEC is to define a common semantic basis for all language constructs. VHDL provides a simulation semantics for each system component. VSPEC provides a declarative, axiomatic semantics for each construct. However, problems tend to arise when migrating information between the two computational models. Modeling inherently operational information declaratively (or vice-versa) simply serves to over complicate the entire model.

Forcing all system component and facet representations into a single semantic model may cause designers to sacrifice useful design abstractions. For example, the abstractions used to model discrete time systems must be sacrificed if analog time is the basic underlying semantics. The same holds true for any single underlying semantic.

The solution to this problem is modeling how facets interact without resorting to a single model. Information should be visible among facets when and where appropriate. It should remain in the facet where it is modeled and be moved directly into the interacting facet without moving through a universal representation. Figure 7 shows graphically the information flow into a unified representation versus information flow between facets.

System facets and their interactions can be viewed theoretically as institutions [6]. Although it is not proposed that facets be implemented as institutions, using this abstraction potentially aids understanding and modeling information.

Each system facet is a category where: (i) objects are model instances in that facet; and (ii) arrows are homomorphisms betweem model instances. To satisfy these requirements, a facet must be a formal system consisting of a language, formal semantics and inference system. Homomorphism is classically defined as theory containment. Specifically, if a homomorphism exists between two objects, then the first is contained in the theory of the second. These characteristics result trivially from category theory definitions.

A category [15] $\mathcal{C}$ is defined as:

1. A collection of **objects**
2. A collection of **morphisms** (represented by arrows)
3. Operations or declarations assigning each arrow f a domain object, d, and co-domain object c. Given $f : a \rightarrow b$ specifies arrow f, $\mathrm{dom}\, f = a$ and $\mathrm{cod}\, f = b$.
4. A composition operator ($\circ$) assigning to each pair of arrows f and g such that $\mathrm{cod}\, f = \mathrm{dom}\, g$ a composite arrow $g \circ f : \mathrm{dom}\, f \rightarrow \mathrm{cod}\, g$ stastifying the associative law:

$$h \circ (g \circ f) = (h \circ g) \circ f$$

5. An identity arrow $id_A : A \rightarrow A$ satisfying the identity law:

$$id_A \circ f = f \wedge f \circ id_A = f$$

```
-- The basic component interface
-- remains the same
entity search is port
  (input: in array of E;
   k: in K;
   output: out E);
end search;

-- A requirements facet containing an
-- axiomatic specification
facet requirements of component is axiomatic
begin
  includes KeyToElement(E,K);
  modifies output;
  sensitive to
    k'event or input'event
  requires true;
  ensures ∀ e: E
    output'post = e iff
      key(e)=k
      and e ∈ input;
end requirements;

-- A requirements facet containing a
-- performance constraint specification
facet constraints of component is peformance
begin
  size ≤ 3 um * 5 um;
  power ≤ 10 mW;
  clock ≤ 50 MHz;
end constraints;
```

```
-- A behavioral facet containing behavioral
-- VHDL
facet function of component is behavioral
  variable i: integer;
begin
  for i in input'range loop
    ...
  end loop;
end function;

-- An architecture facet containing structural
-- VHDL
facet architecture of component is structural
  component sort
    port (list_in: in array of E;
          list_out: out array of E);
  end component;
  component bin_search
    port (list_in: in array of E;
          k: in K;
          e: out E);
  end component;
  signal x: array of E;
begin
  C1: sort port map (input,x);
  C2: bin_search port map (x,k,e);
end architecture;
```

Figure 6. Some potential facets defined using a VHDL-like systems representation.

Homomorphisms between objects within a category represent changes to design instances where correctness is maintained. Relationships between information domains can be represented by treating information domains as categories and interrelationships as functors.

An institution [6] $\mathcal{I}$ is defined as:

1. A category **Sign** of signatures.
2. A functor **Sen** : **Sign** $\rightarrow$ **Set** giving the set of sentences over a given signature.
3. A functor **Mod** : **Sign** $\rightarrow$ **Cat**^{op} giving the variety of models of a given signature
4. A satisfaction relation $\models \subseteq Mod(\Sigma) \times Sen(\Sigma)$ for each Σ in **Sign**

Such that for every morphism $\psi : \Sigma \rightarrow \Sigma'$ in **Sign**, the satisfaction condition:

$$m' \models \psi(e) \Leftrightarrow \psi(m') \models e$$

holds for each m' in **Mod**(Σ') and each e in **Sen**(Σ)

What the institution defines is a link between theorems in one category with theorems in another. The institution enforces a condition that links facets and forces information between them to remain consistent. Thus, if a theorem in one facet changes, appropriate theorems in a linked facet must change to keep information consistent between facets.

Instititions provide the necessary formal framework for a semantic definition. The various facets must be modeled formally as well as the functors regulating interactions. Further, institutions will not form the basis of an efficient implementation. Thus, language structures must be provided that link facets. These language structures can use the institution as their formal basis while providing a more efficient link between two different facets. For example, an institution would be developed that links a requirements facet to a structural facet for a given entity. One possible basis of this institution could be the bisimulation condition defined between VSPEC requirements and VSPEC architectures. Obviously, there is still much work to be done before these concepts can be used to formally describe the relationship between two different facets of a component. However, the institution model appears to be a promising approach.

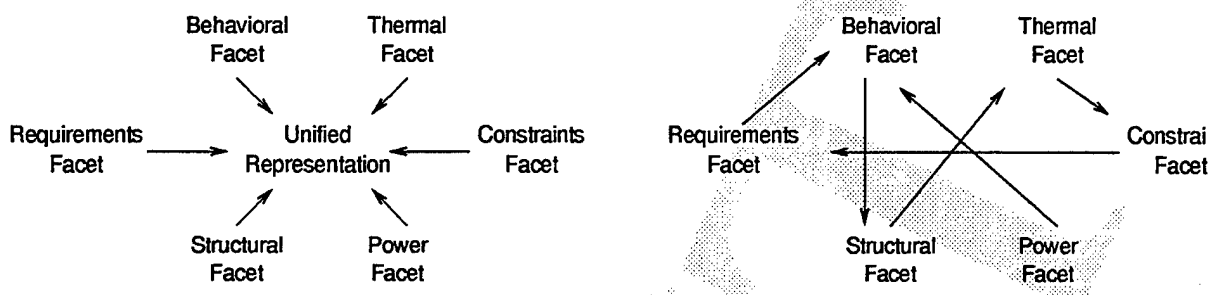


Figure 7. Information flow into a universal representation vs. direct flow between facets.

6. Related Work

Odyssey Research Associates (ORA) is developing Larch/VHDL, an alternative Larch interface language for VHDL [11]. Larch/VHDL is targeted for formal analysis of a VHDL description and ORA is defining a formal semantics for VHDL using LSL. The LSL representations are used in a traditional theorem prover (Penelope, developed for a similar annotation language for Ada [7]) to verify system correctness. Larch/VHDL annotations are added to a specific VHDL description to represent proof obligations for the verification process. This differs from VSPEC's purpose of representing requirements and design decisions at high levels of abstraction. Further, Larch/VHDL provides only a declarative representation of the operational VHDL semantics. However, the interface language defined by ORA does provide a means for defining requirements much like VSPEC's axiomatic component.

Augustin and Luckham's VAL [2] is another attempt to annotate VHDL for requirements modeling. The purpose of a VAL annotation to a VHDL description is to document the design for verification. VAL provides mechanisms for mapping a behavioral description to a structural description. Two VAL/VHDL descriptions of a design can be transformed into a self-checking VHDL program that is simulated to verify that the two descriptions implement the same function. This is once again slightly different than VSPEC's purpose of high level requirements representation. Further, VAL's semantics is operational in that it can be trasformed into VHDL assertions.

The abstract architecture representation capabilities of VSPEC are also fairly closely related to several architecture description languages that have been developed to describe software architectures [5]. Some of the more well known architecture description are UniCon [16], WRIGHT [1] and RAPIDE [12, 13]. Each of these languages allow the definition of components and connectors to define a software architecture. This is very similar to the VHDL notion of a structural architecture.

Allen and Garlan's WRIGHT language is of particular interest when discussing VSPEC because a WRIGHT component is defined with a variant of CSP. Unlike VSPEC's use of CSP to define component synchronization, WRIGHT uses CSP to define component behavior as well. A WRIGHT description consists of a collection of components interacting via instances of connector types. WRIGHT'S CSP descriptions define the sequence of events a component or connector participates in.

7 Conclusions

This paper presented preliminary thoughts on the extension of VHDL to the systems level. First, the systems level design problem was defined as sharing information between system facets. VHDL supports limited multi-facet modeling, but does not provide sufficient flexibility to be called a systems-level design language. Second, VSPEC was presented as a first step towards systems level design. VSPEC adds information facets to VHDL that support modeling requirements, performance constraints and abstract architectures. Finally, initial syntactic and semantic extensions to VHDL were presented that add a `facet` construct to the language and model its semantics using institutions. We believe these extensions would be a first step towards making VHDL more suitable for systems level design.

References

[1] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proc. Sixteenth International Conference on Software Engineering*, pages 71–80, May 1994.

[2] L. Augustin, D. Luckham, B. Gennart, Y. Huh, and A. Stanculescu. *Hardware Design and Simulation in VAL/VHDL*. Kluwer Academic Publishers, Boston, MA, 1991.

[3] P. Baraona and P. Alexander. Representing abstract architectures with axiomatic specifications and activation conditions. In *IEEE Engineering of Computer Based Systems Symposium (ECBS-97)*, March 1997.

[4] P. Baraona, J. Penix, and P. Alexander. VSPEC: A Declarative Requirements Specification Language for VHDL. In J.-M. Berge, O. Levia, and J. Rouillard, editors, *High-Level*

System Modeling: Specification Languages, volume 3 of *Current Issues in Electronic Modeling*, chapter 3, pages 51–75. Kluwer Academic Publishers, Boston, MA, 1995.

[5] D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Eng. and Knowledge Eng.*, volume 2, pages 1–39. World Scientific, New York, 1993.

[6] J. A. Goguen and R. M. Burstall. Introducing institutions. *Lecture Notes in Computer Science*, 164:221–255, 1984.

[7] D. Guaspari. Penelope: An Ada Verification System. In *Proceedingsof Tri-Ada '89*, pages 216–224, Pittsburgh, PA, October 1989.

[8] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[9] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12:576–580,583, 1969.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.

[11] D. Jamsek and M. Bickford. Formal Verification of VHDL Models. Technical Report RL-TR-94-3, Rome Laboratory, Griffiss Air Force Base, NY, March 1994.

[12] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):315–355, April 1995.

[13] D. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[14] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, New York, NY, 1989.

[15] B. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[16] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

DRAFT

APPENDIX N:

Representing Abstract Architectures with Axiomatic Specifications and Activation Conditions*

Phillip Baraona, Perry Alexander
Department of Electrical & Computer Engineering and Computer Science
PO Box 210030 The University of Cincinnati
Cincinnati, OH
{pbaraona,alex}@ececs.uc.edu

Abstract

Evaluating architectural design decisions early in the design process is critical for cost effective design. Formal analysis can provide such evaluation if architectures are defined in a formal way. This paper describes how VSPEC *can be used to formally define an architecture during requirements specification.* VSPEC *is a Larch interface language for* VHDL *that annotates* VHDL *entities using the axiomatic style provided by Larch interface languages. Using* VHDL*'s structural definition support, entities described in this manner are connected to form architectural descriptions. Activation conditions over component inputs define when that component must perform its transform. In this paper, we formally define a* VSPEC *component's state and how component states interact in an architecture. A rudimentary formal semantics for component activation is presented and used to define two potential satisfaction criterion.*

1. Introduction

Architectural design decisions made early in a system's design profoundly affect overall design quality. Unfortunately, architecture decisions are rarely evaluated until late in the design process. Simulation-based design languages such as VHDL [17] do not allow evaluation until complete models exist. Such models include not only architectural decisions, but also component design decisions. For large systems, simulatable models appear late in the design driving up the cost of error correction.

A solution to late architecture evaluation is formal analysis of abstract architectures at the requirements level. An abstract architecture is an inter-connected collection of components where the requirements of each component are specified without defining their implementation. Thus, an abstract architecture describes a class of solutions rather than a single instance. Instead of waiting for a completed system including design detail, formally described abstract architectures can be evaluated when architecture decisions are made. VSPEC [3], a Larch interface language [8] for VHDL [17], is a requirements description language that includes formal architecture definition support.

VSPEC describes the requirements of digital system components using the canonical Larch approach and interconnects component descriptions using VHDL's structural definition features. Each VHDL entity is annotated with a pre- and post-condition to indicate the component's functional requirements. VSPEC-annotated entities are connected together using a VHDL structural architecture to form an abstract architecture. The VHDL architecture indicates interconnection in the traditional manner, but the requirements of each component are defined instead of their implementations. An activation condition can be defined to explicitly indicate when a component should execute. Finally, VSPEC allows a designer to describe non-functional requirements critical in selecting from alternative architecture implementations.

This paper describes VSPEC, concentrating on the language's facilities for describing abstract architectures. Section 2 provides a brief summary of the VSPEC language. Section 3 describes VSPEC abstract architectures, including a definition of the VSPEC state model and a description of how a process algebra (CSP [9]) is used to provide a semantics for the VSPEC activation

*Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615-93-C-1316.

condition. Section 4 discusses how these semantics can be used to verify that an abstract architecture satisfies the specification of the entity. The paper concludes with a discussion of related work.

2. A Brief Summary of VSPEC

VSPEC is a requirements specification language for digital systems. As a requirements specification language, it is used very early in the design process to describe "what" a digital system must do. The operational style of VHDL makes VHDL alone ill-suited for requirements specification. It forces a designer to describe a system by defining a specific design artifact that describes "how" the system behaves. Using VHDL as a requirements specification language forces a designer to deal with unnecessary detail at an early point in the design process.

In contrast to VHDL's operational style, VSPEC allows a designer to declaratively describe a component. A VSPEC description of a sorting component is shown in Figure 1. As with most other Larch interface languages, the **requires** and **ensures** clauses are used to state the pre- and post-conditions of the component. The **sort** component does has a pre-condition of true which means it will function correctly for any set of inputs. The post-condition states that the output contains all the same elements as the input (i.e. **permutation(output'post, input)**) and the output is in order. Any implementation of a sorting component that makes this post-condition true in the next state is a valid implementation of these requirements. More generatlly, given a component with **requires** predicate $I(St)$ and **ensures** predicate $O(St, St'post)$, $f(St)$ is an implementation of the requirements if the following condition holds:

$$\forall s \cdot I(St) \Rightarrow O(St, f(St)) \qquad (1)$$

In addition to allowing a designer to describe "what" a component does, VSPEC also addresses another shortcoming of VHDL: it allows a designer to specify performance constraints in a consistent fashion. The VSPEC **constrained by** clause is used for this purpose. As shown in Figure 1, this clause defines relations over constraint variables. Currently, the defined constraint variables include power consumption, layout area (expressed as a bounding box), heat dissipation, clock speed and pin to pin timing. Constraint theories written in LSL define each constraint type. Users may define their own constraints and theories if desired.

The **state** clause contains a list of variable declarations that define the internal state of a component. These variables maintain state information that may not be recorded by the values of the component's ports. A **state** clause is not needed in the sorting component specification in Figure 1, but an example of this clause can be found in the Move Machine description [3].

The **modifies** clause lists variables, ports and signals whose values may be changed by the entity. Most other Larch interface languages contain a **modifies** clause, and the definition of VSPEC **modifies** clause is very similar to the definitions found in these languages [4, 7, 12]. The **includes** clause is used to include Larch Shared Language definitions in a VSPEC description. The sorts and operators defined in the LSL trait named by the **includes** clause can be used in the VSPEC definition. In this example, the **SortOps** trait defines two predicates: **permutation** and **sorted**.

The **sensitive to** clause plays the same role in a VSPEC definition that sensitivity lists and wait statements play in a VHDL description. It defines when a component is active. The **sensitive to** clause for **sort** in Figure 1 states that the entity activates (and sorts its input) whenever the input changes. The **sensitive to** clause contains a predicate indicating when an entity should begin executing. The next section contains a more precise semantics for the **sensitive to** predicate.

3. Abstract Architectures

VHDL structural architectures composed of VSPEC annotated components specify abstract architectures. The VHDL architecture remains unchanged indicating component instantiation and connections. However, the configuration does not assign an entity/architecture pair to each component instance in the architecture. Instead, the configuration states that each component references an entity with an architecture called **VSPEC**. This signifies that at the current point in the design, the requirements of this component are known (via the VSPEC description) but no implementation has been defined.

Consider the VSPEC description of a **find** component shown in Figure 2a. The output of **find** is the element from the **input** array with the same key as the **k** input. This requirement is represented by **find**'s **ensures** clause. One possible way to meet this requirement is to connect the output of a sorting component to a binary search component as shown in Figure 3. The specification for **sort** is the same as the one in Section 2 while the **bin_search** specification is shown in Figure 2b. The only difference between this structural description of **find** and a VHDL structural description of **find** is the configuration specifies that the VSPEC

```
entity sort is port
  (input: in integer_array;
   output: out integer_array);
  includes SortOps;
  modifies output;
  sensitive to input'event;
  requires true;
  ensures
    permutation(output'post, input) and
    sorted(output'post);
  constrained by
    power <= 5 mW and size <= 3 um * 5 um
    and heat <= 10 mW and clock <= 50 MHz
    and input<->output <= 5 Ms;
end sort;
```

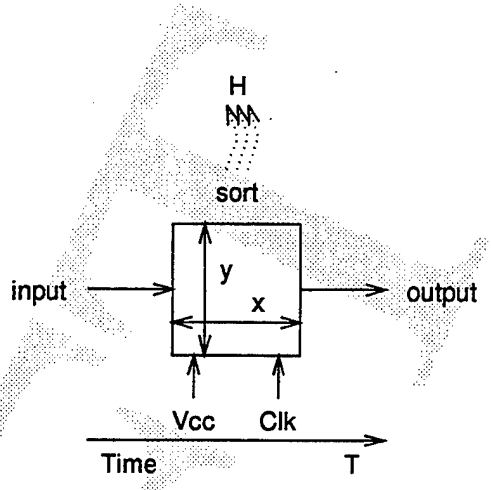


Figure 1. VSPEC description of a sorting component.

descriptions of `sort` and `bin_search` should be used instead of a specific architecture for these two entities. This configuration describes an abstract architecture for the `find` component. Any implementation satisfying the VSPEC requirements of `sort` and `bin_search` may be associated with these `entity` definitions. The abstract architecture for `find` defines a class of solutions with a common structure.

Although a VHDL architecture referencing VSPEC definitions defines components and interconnections, additional information must be added to specify when the VSPEC components activate. In traditional sequential programming, a language construct "executes" following termination of the construct preceding it. For correct execution, a construct's pre-condition must be satisfied when the preceding construct terminates. In hardware systems, components exist simultaneously and behave as independent processes. No predefined execution order exists so there is no means of implicitly determining when a component's pre-condition should hold.

VHDL provides sensitivity lists and `wait` statements to synchronize entity execution and define when a component in a structural architecture is active. VSPEC achieves the same end using the `sensitive to` clause. The `sensitive to` clause contains a predicate called the activation condition that indicates when an entity should begin executing. Effectively, this activation condition defines when a VSPEC annotated entity's precondition must hold. When the `sensitive to` predicate is true, the pre-condition must hold and the implementation must satisfy the post-condition. When the `sensitive to` predicate is false, the entity makes no contribution to the state of the system. In the `find` example, both components activate when any of their input signals change.

Formally, the contribution of the `sensitive to` clause to the transformation specified by VSPEC is easily represented using a traditional process algebra such as CSP [9]. Components become processes and events are defined as the states a component enters. Thus, any VSPEC component can be described by a process that consumes states and generates a process in a new state. To define such state changes, a component state is defined along with a means for combining component states into an architecture state.

The formal VSPEC model of the state of a component is based on Chalin's state model [4, Chapter 6] for LCL. This model partitions the computational state of an LCL description into an environment and a store [19]. The environment maps (variable) identifiers into objects and the store binds objects to the values they contain:

$$Env == Id \rightarrow Obj \quad (2)$$

$$Store == Obj \rightarrow Value \quad (3)$$

Separating the environment and the store in this fashion is common among formal models of program state. In a language such as LCL, a motivating factor for this is to allow multiple names for the same element of memory. For example, two C pointers can obviously reference the same memory location. The program state model above represents this situation by mapping each of these pointers to the same object in the *Env* map.

This partitioning of component state is used in the VSPEC state model. In addition to allowing the correct representation of VHDL `access` types, this partitioning also allows the state of an abstract architecture to be more easily represented. For a single VSPEC-specified

```
entity find is port
  (input: in element_array;
   k: in keytype;
   output: out element);
  includes Element(element,keytype,
                   element_array);
  modifies output;
  sensitive to
    input'event or k'event;
  requires true;
  ensures forall (e : element)
    (output = e implies
       (e.key = k
       and elem_of(e,input)));
  constrained by
    power <= 5 mW
    and size <= 3 um * 5 um
    and k<->output <= 5 Ms
    and heat <= 10 mW
    and clock <= 50 MHz;
end find;
```

```
entity bin_search is
  port (input: buffer element_array;
        k: in integer;
        value: out element);
  modifies value;
  sensitive to
    input'event or k'event;
  requires sorted(input);
  ensures output = e iff (e.key=k and
          element_of(e,input));
  constrained by
    power <= 1 mW and
    size <= 1 um * 2 um;
end bin_search;
```

Figure 2. VSPEC descriptions of `find` and `binary search` components.

component, *Env* contains a map from each port and state variable in the VSPEC description to an object. *Store* maps each of these objects to their current value. We call this the *abstract state* of the VSPEC component.

When VSPEC components are connected together to form an abstract architecture, the elements of *Env* and *Store* are slightly different. The *Store* contains objects for each port in the architecture's entity, for each signal in the architecture and for the state variables of each component in the architecture. The *Env* maps each of these three types of elements to the proper object, but it also maps the ports of each architecture component to the object that represents the architecture signal the port is connected to. We call the state model of an abstract architecture the *concrete state* of the component.

In the simple two component example of Figure 4, the abstract state of `system`, `A` and `B` are:

$$
\begin{aligned}
Env_{system} &= \{sys_in \mapsto obj_{sys_in}, \\
&\quad\ sys_out \mapsto obj_{sys_out}\} \\
Store_{system} &= \{obj_{sys_in} \mapsto v_{sys_in}, \\
&\quad\ obj_{sys_out} \mapsto v_{sys_out}\} \\
Env_A &= \{x \mapsto obj_x, y \mapsto obj_y\} \\
Store_A &= \{obj_x \mapsto v_x, obj_y \mapsto v_y\} \\
Env_B &= \{w \mapsto obj_w, z \mapsto obj_z\} \\
Store_B &= \{obj_w \mapsto v_w, obj_z \mapsto v_z\}
\end{aligned}
$$

The concrete state of the `struct` architecture is:

$$
\begin{aligned}
Env_{struct_{system}} &= \{sys_in \mapsto obj_{sys_in}, \\
&\quad\ sys_out \mapsto obj_{sys_out}, \\
&\quad\ c \mapsto obj_c, x \mapsto obj_{sys_in}, \\
&\quad\ y \mapsto obj_c, w \mapsto obj_c, \\
&\quad\ z \mapsto obj_{sys_out}\} \\
Store_{struct_{system}} &= \{obj_{sys_in} \mapsto v_{sys_in}, \\
&\quad\ obj_{sys_out} \mapsto v_{sys_out}, obj_c \mapsto v_c\}
\end{aligned}
$$

Notice that `x`, `y`, `w` and `z` now map to the objects containing the signal values the component ports are connected to.

The semantics of a VSPEC entity are defined by a CSP process that defines the sequence of states the entity passes through. Let C be an entity with `sensitive to`, `requires` and `ensures` predicates $S(St)$, $I(St)$ and $O(St, St'post)$, respectively. The process defining C in any state r is:

$$C_r = r : \Psi \rightarrow C_{r'post} \tag{4}$$

where $\Psi = \{t : T_C | S(t)\}$ is the set of states that satisfy C's activation condition and P_x is the process P in some state x. $O(r, r'post)$ must hold to assure the transformation's correctness. Thus, when an external force changes the abstract state to one that satisfies the entity's activation condition (r in Equation 4), the process will consume r and behave like $C_{r'post}$. A trace of the process defined by a VSPEC entity is a sequence

```
architecture structure of find is
  component sorter
    port (input: in element_array;
          output: out element_array);
  end component;
  component searcher
    port (input: in element_array;
          key: in integer;
          value: out element);
  end component;
  signal y: element_array;
begin
  b1: sorter port map(input,y);
  b2: searcher port map(y,k,output);
end structure;
```

```
configuration test_vspec of find is
  for structure
    for b1:sorter use entity
      work.sort(VSPEC);
    end for;
    for b2:searcher use entity
      work.bin_search(VSPEC);
    end for;
  end for;
end test_struct;
```

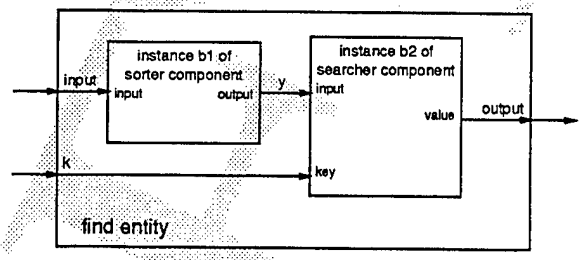


Figure 3. A VSPEC abstract architecture representation of the `find` component.

of abstract states the entity enters. Each of these states satisfy C's activation condition. Thus, the alphabet of C is equal to Ψ.

If $f(St)$ implements the requirements specified by $I(St)$ and $O(St, St'post)$ (i.e. $f(St)$ satisfies Equation 1), Equation 4 can be re-written as:

$$C_r = r : \Psi \rightarrow C_{f(r)} \tag{5}$$

In this situation, the process consumes r and f is applied to r to generate a new abstract state. The entity then behaves like the process defined by $C_{f(r)}$.

CSP's concurrency operator combines component processes to define the behavior of a VSPEC architecture. Let $C_1, C_2, ..., C_n$ be the processes represented by Equation 4 or 5 for the set of VSPEC component instances in architecture $\mathcal{A}$. The process representing architecture $\mathcal{A}$ is:

$$\mathcal{A} = C_1 \parallel C_2 \parallel ... \parallel C_n \tag{6}$$

When the current state satisfies some component's activation condition, the component performs its specified transformation to its abstract state. This change is propagated to the concrete state of the architecture where the activation condition of another component may be satisfied. This causes the process to repeat until the system changes to a concrete state where no component's activation condition is satisfied. The system then waits until some external source changes the concrete state to one that activates some component in the architecture to start the process again.

In the CSP model of a VSPEC process, this notion can be understood by examining the possible traces of $\mathcal{A}$ from Equation 6. Hoare [9] defines traces over parallel composition, $traces(C_1 \parallel C_2)$, as:

$$\{t | (t \upharpoonright \alpha C_1) \in traces(C_1) \\ \wedge (t \upharpoonright \alpha C_2) \in traces(C_2) \\ \wedge t \in (\alpha C_1 \cup \alpha C_2)^*\}$$

Thus, the traces of a parallel composition of components are all traces that when restricted to the alphabet of each component yield a trace of that component. Furthermore, traces of $C_1 \parallel C_2$ only contain events from the alphabet of either components. Thus, every trace of $\mathcal{A}$ contains only states that satisfy the activation condition of at least one component in $\mathcal{A}$.

If $\mathcal{A}$ enters a state where none of its component's activation condition is true, it will wait for a change on one of its input ports. Sequences in $traces(\mathcal{A})$ contain only states that activate a component of $\mathcal{A}$ so the process representing $\mathcal{A}$ only consumes those states. However, a change to a component's input port also causes a state change and inactive components must wait for events from external sources to initiate activation. $Traces(\mathcal{A})$ is not strictly the set of all states a component may enter, but the set of all states a component enters from active states.

4. System Verification

This section describes how the CSP semantics of a VSPEC abstract architecture can be used to verify that an abstract architecture for an entity satisfies the VSPEC specification of the entity. Many satisfaction

```
entity A is port
  (x : in integer;
   y : out integer);
  requires I_A(x);
  ensures O_A(x, y'post);
  modifies y;
end A;

entity B is port
  (w : in integer;
   z : out integer);
  requires I_B(w);
  ensures O_B(w, z'post);
  modifies z;
end B;

entity system is port
  (sys_in : in integer;
   sys_out : out integer);
end system;
```

```
architecture struct of system is
component A
  port (x : in integer;
        y : out integer);
end component;
component B
  port (w : in integer;
        z : out integer);
end component;
signal c;

begin
  c1: A port map(sys_in,c);
  c2: B port map(c,sys_out);
end struct;
```

Figure 4. Example of two entities connected serially.

criteria could be specified and checked. Here, two examples are considered: (1) weak bisimulation; and (2) trace equivalence. Weak bisimulation will evaluate the final state of a halting system. Trace equivalence will look at traces from systems that do not halt.

Satisfaction criteria will be evaluated by comparing the abstract states from the problem definition with concrete states of the abstract architecture. To make this comparison possible, an abstraction function that maps concrete states to their abstract equivalent must be defined. We call this function *abs* and note that a concrete state c is equivalent to an abstract state a if and only if $abs(c) = a$.

The most traditional correctness criterion used to verify an abstract architecture implements its specification is weak bisimulation [15]. A weak bisimulation (or simply bisimulation) condition holds when a sequence of states in the concrete model produces a desired single state change specified by the abstract model (see Figure 5). Only the first and last state of the concrete state sequence are significant. The specific state sequence leading from the initial concrete state to the final concrete state is ignored.

Equation 7 is a weak bisimulation correctness obligation for showing architecture $\mathcal{A}$ satisfies a single abstract state change specification. Here, $\Psi_{\mathcal{A}}$ is the set of concrete states where the activation condition of at least one component in $\mathcal{A}$ is true. The obligation states that for concrete state traces starting in a state whose abstract projection satisfies the abstract specification's pre-condition, either the abstract projection of the final process state in the trace satisfies the component post-condition or the process can consume the state and continue.

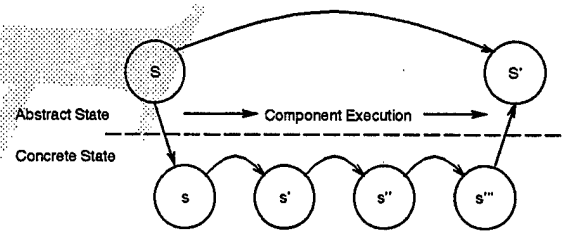


Figure 5. Concrete state changes associated with a single abstract state change.

$$\forall \tau : traces(\mathcal{A}) \cdot I(abs(\tau_0)) \wedge \mathcal{A}/\tau = \mathcal{A}_s \Rightarrow (O(abs(\tau_0), abs(s)) \vee s \in \Psi_{\mathcal{A}})$$

For systems with clearly defined halting or pausing points, Equation 7 is an appropriate correctness criterion. However, many systems run continuously. Their states are observable, but there is no notion of pausing or halting to synchronize abstract state comparison. To formulate the correctness criterion for these types of systems, a concept similar to bisimulation is applied to sequences of states rather than a single state change.

Traces can be derived from the abstract requirement specification by defining process R in state S as $R_S = S : \Psi \rightarrow R_{S'}$ in the same manner as the con-

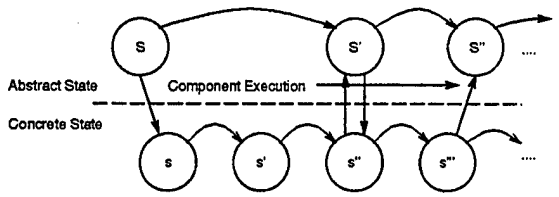


Figure 6. Concrete state changes associated with multiple abstract state changes.

crete requirements. Such traces are exactly one event long when a single state change is defined. However, if the resulting state satisfies the component's activation condition, then the process will continue to consume states. Thus, $traces(R_S)$ is the set of finite abstract state sequences defined for process R. With this, traces through both the abstract requirements and concrete specification are defined.

The image of a trace with respect to an abstraction function, *abs*, is simply the abstraction function applied to each trace element, $image(\langle e_0, e_1, \ldots, e_n \rangle) = \langle abs(e_0), abs(e_1), \ldots, abs(e_n) \rangle$. The *reduce* function eliminates invisible state changes by replacing adjacent equivalent states in a trace with a single state. For example, $reduce(\langle a, b, a, a, c, c, c \rangle) = \langle a, b, a, c \rangle$.

A concrete specification is correct with respect to reduced abstract equivalence if:

$$\forall\, t : traces(P) \cdot reduce(t) \in traces(R). \qquad (7)$$

In this case, an architecture specification is correct if every trace of concrete states can be reduced to a legal trace of abstract states. Reducing the state sequence removes concrete state changes that are not observable in the external state. It should be noted that the component semantics thus far specifies only liveness properties (what the system must do) and largely ignores safety properties (what the system must not do) [11]. The weak bisimulation semantic specifies only characteristics of the resultant state and by definition ignores characteristics of intermediate states. This should not be viewed as a fatal flaw because this is precisely what traditional block diagrams define. Some methodologies may extend the block diagram approach to include safety properties, but the traditional diagram specifies only what must happen and when it must happen.

5. Related Work

Odyssey Research Associates (ORA) is developing Larch/VHDL, an alternative Larch interface language for VHDL [10]. Larch/VHDL is targeted for formal analysis of a VHDL description and ORA is defining a formal semantics for VHDL using LSL. The LSL representations are used in a traditional theorem prover (Penelope, developed for a similar annotation language for Ada [6])to verify system correctness. Larch/VHDL annotations are added to a specific VHDL description to represent proof obligations for the verification process. This differs from VSPEC's purpose of representing requirements and design decisions at high levels of abstraction.

Augustin and Luckham's VAL [2] is another attempt to annotate VHDL. The purpose of a VAL annotation to a VHDL description is to document the design for verification. VAL provides mechanisms for mapping a behavioral description to a structural description. Two VAL/VHDL descriptions of a design can be transformed into a self-checking VHDL program that is simulated to verify that the two descriptions implement the same function. This is once again slightly different than VSPEC's purpose of high level requirements representation.

The abstract architecture representation capabilities of VSPEC are also fairly closely related to several architecture description languages that have been developed to describe software architectures [5]. Some of the more well known architecture description are UniCon [18], WRIGHT [1] and RAPIDE [13, 14]. Each of these languages allow the definition of components and connectors to define a software architecture. This is very similar to the VHDL notion of a structural architecture.

Allen and Garlan's WRIGHT language is of particular interest when discussing VSPEC because a WRIGHT component is defined with a variant of CSP. Unlike VSPEC's use of CSP to define component synchronization, WRIGHT uses CSP to define component behavior as well. A WRIGHT description consists of a collection of components interacting via instances of connector types. WRIGHT's CSP descriptions define the sequence of events a component or connector participates in.

6. Conclusions

This paper presented VSPEC, a requirements specification language for VHDL, emphasizing VSPEC architecture representation. A VSPEC specification describes the pre-condition, post-condition, performance constraints and activation condition of a VHDL entity. When the activation condition is true, the entity's precondition must hold and the entity is responsible for making its post-condition hold in the next state. The semantics of a single component VSPEC specification is based on the canonical Larch axiomatic approach while CSP is used to define the semantics of an architecture

of components. Two satisfaction criterion used to verify that an architecture is a refinement of requirements specification were discussed here: weak bisimulation and trace equality. Weak bisimulation evaluated an architecture's halting state with respect to a requirements specification. Trace equality compared state traces from systems that do not halt. These mechanisms allow an architectural description to be formally analyzed at the requirements level.

At the present time, the first version of the language definition is complete. A VSPEC parser that type checks expressions by calling an LSL parser has been implemented. Constraint theories for the five basic constraints (power, area, heat dissipation, clock speed and pin to pin timing) have been developed. The formal semantics of a single component VSPEC specification based on the canonical Larch approach is complete as is the first cut at the semantics of an abstract architecture using CSP. Several specifications using these techniques have been developed, but further investigation into architecture semantics is needed.

The main area of future work for VSPEC is to refine the semantics of an abstract architecture of VSPEC components. The CSP semantics presented in this paper are useful, but we may investigate using a different process algebra such as CCS [16] to describe architectures. The main reason for this is that weak bisimulation was originally formulated using CCS and it may be more natural to reason about weak bisimulation using this process algebra.

One of the primary goals of this research is to provide a mechanism that allows the affects of architecture decisions to be evaluated earlier in the design process. VSPEC accomplishes this goal by allowing components in an architecture to be described using a traditional axiomatic specification and formally modeling the interactions between components using a process algebra. This approach allows architecture decisions to be evaluated at the requirements level which should improve overall design quality.

References

[1] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proc. Sixteenth International Conference on Software Engineering*, pages 71–80, May 1994.

[2] L. Augustin, D. Luckham, B. Gennart, Y. Huh, and A. Stanculescu. *Hardware Design and Simulation in VAL/VHDL*. Kluwer Academic Publishers, Boston, MA, 1991.

[3] P. Baraona, J. Penix, and P. Alexander. VSPEC: A Declarative Requirements Specification Language for VHDL. In J.-M. Berge, O. Levia, and J. Rouillard, editors, *High-Level System Modeling: Specification Languages*, volume 3 of *Current Issues in Electronic Modeling*, chapter 3, pages 51–75. Kluwer Academic Publishers, Boston, MA, 1995.

[4] P. Chalin. *On the Language Design and Semantic Foundation of LCL, a Larch/C Interface Specification Language*. PhD thesis, Concordia University, Department of Computer Science, Montreal, Quebec, Canada, December 1995.

[5] D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Eng. and Knowledge Eng.*, volume 2, pages 1–39. World Scientific, New York, 1993.

[6] D. Guaspari. Penelope: An Ada Verification System. In *Proceedingsof Tri-Ada '89*, pages 216–224, Pittsburgh, PA, October 1989.

[7] J. V. Guttag and J. J. Horning. Introduction to LCL, A Larch/C Interface Language. Technical Report 74, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, July 1991.

[8] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[9] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.

[10] D. Jamsek and M. Bickford. Formal Verification of VHDL Models. Technical Report RL-TR-94-3, Rome Laboratory, Griffiss Air Force Base, NY, March 1994.

[11] L. Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[12] G. T. Leavens. Larch/C++ reference manual. Available at: `ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz`, 1995.

[13] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):315–355, April 1995.

[14] D. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[15] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[16] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, New York, NY, 1989.

[17] D. Perry. *VHDL*. McGraw-Hill, New York, NY, 1st edition, 1991.

[18] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[19] R. Tennent. *Principles of Programming Languages*. Computer Science Series. Prenitce-Hall International, 1981.

APPENDIX 0:

Formal Representations for Abstract System Evaluation*

Perry Alexander
Department of Electrical & Computer Engineering and Computer Science
PO Box 210030 The University of Cincinnati
Cincinnati, OH
alex@ececs.uc.edu

Abstract

Evaluating design decisions early in the design process is critical for cost effective design. Formal analysis can provide such evaluation if architectures are defined in a formal way. VSPEC *is a Larch interface language for* VHDL *that annotates* VHDL *entities using the axiomatic style provided by Larch interface languages. Using* VHDL*'s structural definition support, entities described in this manner can be connected to form architectural descriptions. Activation conditions over component inputs define when the component must perform its transform. In this paper, we provide a simple introduction to* VSPEC *and its mechanisms for describing systems architectures.*

1 Introduction

Design decisions made early in a system's design profoundly affect overall design quality. Unfortunately, such decisions are rarely evaluated until late in the design process. Simulation-based design languages such as VHDL [10] do not allow evaluation until complete models exist. Such models include not only abstract decisions, but also low level component design decisions. For large systems, simulatable models appear late in the design increasing the cost of error correction.

A solution to late evaluation is formal analysis at the requirements level. Formal representation of requirements and abstract architectures supports analysis of incomplete systems at high abstraction levels. Furthermore, formalisms provide some guarantee of rigor in representation and correctness in analysis. Abstract architectures support representation and analysis of requirements partitioning attempts and architecture level design decisions.

Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615–93–C–1316.

An abstract architecture is an inter-connected collection of components where the requirements of each component are specified without defining their implementation. Thus, an abstract architecture describes a class of solutions rather than a single instance. Instead of waiting for a completed system including design detail, formally described abstract architectures can be evaluated when architecture decisions are made. VSPEC [1, 2], a Larch interface language [4, 6] for VHDL [10], is a requirements description language that includes formal architecture definition support.

VSPEC describes the requirements of digital system components using the canonical Larch approach and interconnects component descriptions using VHDL's structural definition features. Each VHDL entity is annotated with a pre- and post-condition to indicate the component's functional requirements. VSPEC-annotated entities are connected together using a VHDL structural architecture to form an abstract architecture. The VHDL architecture indicates interconnection in the traditional manner, but the requirements of each component are defined instead of their implementations. An activation condition can be defined to explicitly indicate when a component should execute. Finally, VSPEC allows a designer to describe non-functional requirements critical in selecting from alternative architecture implementations.

2 A Brief Summary of VSPEC

VSPEC is a requirements specification language for digital systems. As a requirements specification language, it is used very early in the design process to describe "what" a digital system must do. The operational style of VHDL makes VHDL alone ill-suited for requirements specification. It forces a designer to describe a system by defining a specific design artifact that describes "how" the system behaves. Using VHDL as a requirements specification language forces a designer to deal with unnecessary detail at an early point in the design process.

In contrast to VHDL's operational style, VSPEC allows a designer to declaratively describe a component. A VSPEC description of a sorting component is shown in Figure 1. As with most other Larch interface languages, the **requires** and **ensures** clauses are used to state the pre- and post-conditions of the component. The **sort** component does has a precondition of true which means it will function correctly for any set of inputs. The post-condition states that the output contains all the same elements as the input (i.e. `permutation(output'post, input)`) and the output is in order. Any implementation of a sorting component that makes this post-condition true in the next state is a valid implementation of these requirements. More generatlly, given a component with **requires** predicate $I(St)$ and **ensures** predicate $O(St, St'post)$[1], $f(St)$ is an implementation of the requirements if the following condition holds:

$$\forall s \bullet I(St) \Rightarrow O(St, f(St)) \qquad (1)$$

In addition to allowing a designer to describe "what" a component does, VSPEC also addresses another shortcoming of VHDL: it allows a designer to specify performance constraints in a consistent fashion. The VSPEC **constrained by** clause is used for this purpose. As shown in Figure 1, this clause defines relations over constraint variables. Currently, the defined constraint variables include power consumption, layout area (expressed as a bounding box), heat dissipation, clock speed and pin to pin timing. Constraint theories written in LSL define each constraint type. Users may define their own constraints and theories if desired.

The **state** clause contains a list of variable declarations that define the internal state of a component. These variables maintain state information that may not be recorded by the values of the component's ports. A **state** clause is not needed in the sorting component specification in Figure 1.

The **modifies** clause lists variables, ports and signals whose values may be changed by the entity. Most other Larch interface languages contain a **modifies** clause, and the definition of VSPEC **modifies** clause is very similar to the definitions found in these languages [3, 5, 8]. The **includes** clause is used to include Larch Shared Language definitions in a VSPEC description. The sorts and operators defined in the LSL trait named by the **includes** clause can be used in the VSPEC definition. In this example, the `SortOps` trait defines two predicates: **permutation** and **sorted**.

The **sensitive to** clause plays the same role in a VSPEC definition that sensitivity lists and wait statements play in a VHDL description. It defines when a component is active. The **sensitive to** clause for **sort** in Figure 1 states that the entity activates (and sorts its input) whenever the input changes. The **sensitive to** clause contains a predicate indicating when an entity should begin executing. The next section contains a more precise semantics for the **sensitive to** predicate.

3 Abstract Architectures

VHDL structural architectures composed of VSPEC annotated components specify abstract architectures. The VHDL architecture remains unchanged indicating component instantiation and connections. However, the configuration does not assign an entity/architecture pair to each component instance in the architecture. Instead, the configuration states that each component references an entity with an architecture called VSPEC. This signifies that at the current point in the design, the requirements of this component are known (via the VSPEC description) but no implementation has been defined.

Consider the VSPEC description of a **find** component shown in Figure 2a. The output of **find** is the element from the **input** array with the same key as the **k** input. This requirement is represented by **find**'s **ensures** clause. One possible way to meet this requirement is to connect the output of a sorting component to a binary search component as shown in Figure 3. The specification for **sort** is the same as the one in Section 2 while the **bin_search** specification is shown in Figure 2b. The only difference between this structural description of **find** and a VHDL structural description of **find** is the configuration specifies that the VSPEC descriptions of **sort** and **bin_search** should be used instead of a specific architecture for these two entities. This configuration describes an abstract architecture for the **find** component. Any implementation satisfying the VSPEC requirements of **sort** and **bin_search** may be associated with these **entity** definitions. The abstract architecture for **find** defines a class of solutions with a common structure.

Although a VHDL architecture referencing VSPEC definitions defines components and interconnections, additional information must be added to specify when the VSPEC components activate. In traditional sequential programming, a language construct "executes" following termination of the construct pre-

[1] The $St'post$ notation references the value of St in the state after the transformation described by the entity is performed. This is analogous to the *variable'* notation of LCL [3, 5]

```
entity sort is port
  (input: in integer_array;
   output: out integer_array);
  includes SortOps;
  modifies output;
  sensitive to input'event;
  requires true;
  ensures
    permutation(output'post, input) and
    sorted(output'post);
  constrained by
    power <= 5 mW and size <= 3 um * 5 um
    and heat <= 10 mW and clock <= 50 MHz
    and input<->output <= 5 Ms;
end sort;
```

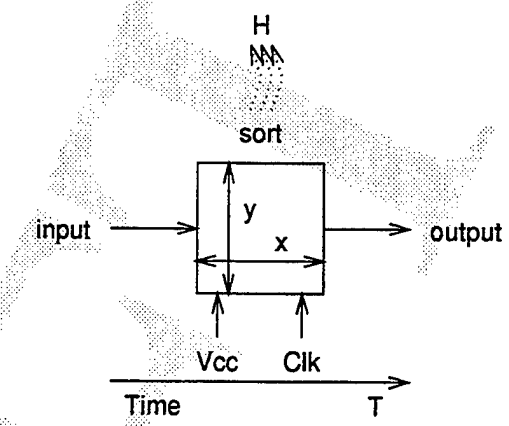


Figure 1: VSPEC description of a sorting component.

ceding it. For correct execution, a construct's pre-condition must be satisfied when the preceding construct terminates. In hardware systems, components exist simultaneously and behave as independent processes. No predefined execution order exists so there is no means of implicitly determining when a component's pre-condition should hold.

VHDL provides sensitivity lists and `wait` statements to synchronize entity execution and define when a component in a structural architecture is active. VSPEC achieves the same end using the `sensitive to` clause. The `sensitive to` clause contains a predicate called the activation condition that indicates when an entity should begin executing. Effectively, this activation condition defines when a VSPEC annotated entity's precondition must hold. When the `sensitive to` predicate is true, the pre-condition must hold and the implementation must satisfy the post-condition. When the `sensitive to` predicate is false, the entity makes no contribution to the state of the system. In the `find` example, both components activate when any of their input signals change.

Formally, the contribution of the `sensitive to` clause to the transformation specified by VSPEC is easily represented using a traditional process algebra such as CSP [7]. Components become processes and events are defined as the states a component enters. Thus, any VSPEC component can be described by a process that consumes states and generates a process in a new state. To define such state changes, a component state is defined along with a means for combining component states into an architecture state.

The formal VSPEC model of the state of a component is based on Chalin's state model [3, Chapter 6] for LCL. This model partitions the computational state of an LCL description into an environment and a store [11]. The environment maps (variable) identifiers into objects and the store binds objects to the values they contain:

$$Env == Id \rightarrow Obj \quad (2)$$

$$Store == Obj \rightarrow Value \quad (3)$$

Separating the environment and the store in this fashion is common among formal models of program state. In a language such as LCL, a motivating factor for this is to allow multiple names for the same element of memory. For example, two C pointers can obviously reference the same memory location. The program state model above represents this situation by mapping each of these pointers to the same object in the *Env* map.

This partitioning of component state is used in the VSPEC state model. In addition to allowing the correct representation of VHDL `access` types, this partitioning also allows the state of an abstract architecture to be more easily represented. For a single VSPEC-specified component, *Env* contains a map from each port and state variable in the VSPEC description to an object. *Store* maps each of these objects to their current value. We call this the *abstract state* of the VSPEC component.

When VSPEC components are connected together to form an abstract architecture, the elements of *Env* and *Store* are slightly different. The *Store* contains objects for each port in the architecture's entity, for each signal in the architecture and for the state variables of each component in the architecture. The *Env* maps each of these three types of elements to the proper

```
entity find is port
  (input: in element_array;
   k: in keytype;
   output: out element);
  includes Element(element,keytype,
                   element_array);
  modifies output;
  sensitive to
    input'event or k'event;
  requires true;
  ensures forall (e : element)
    (output = e implies
       (e.key = k
       and elem_of(e,input)));
  constrained by
    power <= 5 mW
    and size <= 3 um * 5 um
    and k<->output <= 5 Ms
    and heat <= 10 mW
    and clock <= 50 MHz;
end find;
```

(a.)

```
entity bin_search is
  port (input: buffer element_array;
        k: in integer;
        value: out element);
  modifies value;
  sensitive to
    input'event or k'event;
  requires sorted(input);
  ensures output = e iff (e.key=k and
          element_of(e,input));
  constrained by
    power <= 1 mW and
    size <= 1 um * 2 um;
end bin_search;
```

(b.)

Figure 2: VSPEC descriptions of `find` and `binary search` components.

object, but it also maps the ports of each architecture component to the object that represents the architecture signal the port is connected to. We call the state model of an abstract architecture the *concrete state* of the component.

In the simple two component example of Figure 4, the abstract state of **system**, **A** and **B** are:

$$
\begin{aligned}
Env_{system} &= \{sys_in \mapsto obj_{sys_in}, \\
&\quad sys_out \mapsto obj_{sys_out}\} \\
Store_{system} &= \{obj_{sys_in} \mapsto v_{sys_in}, \\
&\quad obj_{sys_out} \mapsto v_{sys_out}\} \\
Env_A &= \{x \mapsto obj_x, y \mapsto obj_y\} \\
Store_A &= \{obj_x \mapsto v_x, obj_y \mapsto v_y\} \\
Env_B &= \{w \mapsto obj_w, z \mapsto obj_z\} \\
Store_B &= \{obj_w \mapsto v_w, obj_z \mapsto v_z\}
\end{aligned}
$$

The concrete state of the **struct** architecture is:

$$
\begin{aligned}
Env_{struct_{system}} &= \{sys_in \mapsto obj_{sys_in}, \\
&\quad sys_out \mapsto obj_{sys_out}, \\
&\quad c \mapsto obj_c, x \mapsto obj_{sys_in}, \\
&\quad y \mapsto obj_c, w \mapsto obj_c, \\
&\quad z \mapsto obj_{sys_out}\} \\
Store_{struct_{system}} &= \{obj_{sys_in} \mapsto v_{sys_in}, \\
&\quad obj_{sys_out} \mapsto v_{sys_out}, \\
&\quad obj_c \mapsto v_c\}
\end{aligned}
$$

Notice that **x**, **y**, **w** and **z** now map to the objects containing the signal values the component ports are connected to.

The semantics of a VSPEC entity are defined by a CSP process that defines the sequence of states the entity passes through. Let C be an entity with **sensitive to**, **requires** and **ensures** predicates $S(St)$, $I(St)$ and $O(St, St'post)$, respectively. The process defining C in any state r is:

$$C_r = r : \Psi \rightarrow C_{r'post} \tag{4}$$

where $\Psi = \{t : T_C | S(t)\}$ is the set of states that satisfy C's activation condition and P_x is the process P in some state x. $O(r, r'post)$ must hold to assure the transformation's correctness. Thus, when an external force changes the abstract state to one that satisfies the entity's activation condition (r in Equation 4), the process will consume r and behave like $C_{r'post}$. A trace of the process defined by a VSPEC entity is a sequence of abstract states the entity enters. Each of these states satisfy C's activation condition. Thus, the alphabet of C is equal to Ψ.

```
architecture structure of find is
  component sorter
    port (input: in element_array;
          output: out element_array);
  end component;
  component searcher
    port (input: in element_array;
          key: in integer;
          value: out element);
  end component;
  signal y: element_array;
begin
  b1: sorter port map(input,y);
  b2: searcher port map(y,k,output);
end structure;
```

```
configuration test_vspec of find is
  for structure
    for b1:sorter use entity
      work.sort(VSPEC);
    end for;
    for b2:searcher use entity
      work.bin_search(VSPEC);
    end for;
  end for;
end test_struct;
```

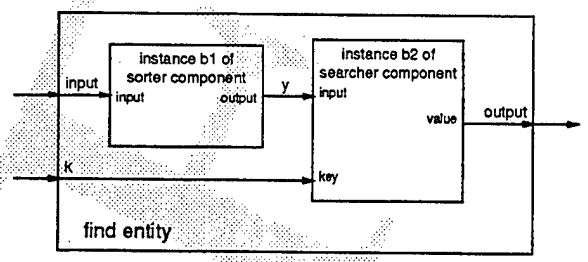


Figure 3: A VSPEC abstract architecture representation of the find component.

If $f(St)$ implements the requirements specified by $I(St)$ and $O(St, St'post)$ (i.e. $f(St)$ satisfies Equation 1), Equation 4 can be re-written as:

$$C_r = r : \Psi \rightarrow C_{f(r)} \tag{5}$$

In this situation, the process consumes r and f is applied to r to generate a new abstract state. The entity then behaves like the process defined by $C_{f(r)}$.

CSP's concurrency operator combines component processes to define the behavior of a VSPEC architecture. Let $C_1, C_2, ..., C_n$ be the processes represented by Equation 4 or 5 for the set of VSPEC component instances in architecture $\mathcal{A}$. The process representing architecture $\mathcal{A}$ is:

$$\mathcal{A} = C_1 \parallel C_2 \parallel ... \parallel C_n \tag{6}$$

When the current state satisfies some component's activation condition, the component performs its specified transformation to its abstract state. This change is propagated to the concrete state of the architecture where the activation condition of another component may be satisfied. This causes the process to repeat until the system changes to a concrete state where no component's activation condition is satisfied. The system then waits until some external source changes the concrete state to one that activates some component in the architecture to start the process again.

In the CSP model of a VSPEC process, this notion can be understood by examining the possible traces of $\mathcal{A}$ from Equation 6. Hoare [7] defines traces over parallel composition, $traces(C_1 \parallel C_2)$, as:

$$\begin{aligned} traces(C_1 \parallel C_2) \quad = \quad & \{t | (t \upharpoonright \alpha C_1) \in traces(C_1) \\ & \wedge (t \upharpoonright \alpha C_2) \in traces(C_2) \\ & \wedge t \in (\alpha C_1 \cup \alpha C_2)^*\} \end{aligned}$$

Thus, the traces of a parallel composition of components are all traces that when restricted to the alphabet of each component yield a trace of that component. [2] Furthermore, traces of $C_1 \parallel C_2$ only contain events from the alphabet of either components. Thus, every trace of $\mathcal{A}$ contains only states that satisfy the activation condition of at least one component in $\mathcal{A}$.

If $\mathcal{A}$ enters a state where none of its component's activation condition is true, it will wait for a change on one of its input ports. Sequences in $traces(\mathcal{A})$ contain only states that activate a component of $\mathcal{A}$ so the process representing $\mathcal{A}$ only consumes those states. However, a change to a component's input port also causes a state change and inactive components must wait for events from external sources to initiate activation. $Traces(\mathcal{A})$ is not strictly the set of all states a component may enter, but the set of all states a component enters from active states.

4 Conclusions

This paper presented a basic introduction to VSPEC, a requirements specification language for VHDL. A VSPEC specification describes the pre-condition, post-condition, performance constraints and activation

[2] Recall that in CSP [7], $t \upharpoonright \alpha P$ restricts the trace t to contain only events that appear in the alphabet of P.

```
entity A is port
  (x : in integer;
   y : out integer);
  requires I_A(x);
  ensures O_A(x, y'post);
  modifies y;
end A;

entity B is port
  (w : in integer;
   z : out integer);
  requires I_B(w);
  ensures O_B(w, z'post);
  modifies z;
end B;

entity system is port
  (sys_in : in integer;
   sys_out : out integer);
end system;
```

```
architecture struct of system is
component A
  port (x : in integer;
        y : out integer);
end component;
component B
  port (w : in integer;
        z : out integer);
end component;
signal c;

begin
  c1: A port map(sys_in,c);
  c2: B port map(c,sys_out);
end struct;
```

Figure 4: Example of two entities connected serially.

condition of a VHDL entity. When the activation condition is true, the entity's pre-condition must hold and the entity is responsible for making its post-condition hold in the next state. The semantics of a single component VSPEC specification is based on the canonical Larch axiomatic approach while CSP is used to define the semantics of an architecture of components.

References

[1] ALEXANDER, P., BARAONA, P., AND PENIX, J. Using Declarative Specifications and Case-Based Planning for System Synthesis. *Concurrent Engineering: Research and Applications 2*, 4 (1994).

[2] BARAONA, P., PENIX, J., AND ALEXANDER, P. VSPEC: A Declarative Requirements Specification Language for VHDL. In *High-Level System Modeling: Specification Languages*, J.-M. Berge, O. Levia, and J. Rouillard, Eds., vol. 3 of *Current Issues in Electronic Modeling*. Kluwer Academic Publishers, Boston, MA, 1995, ch. 3, pp. 51–75.

[3] CHALIN, P. *On the Language Design and Semantic Foundation of LCL, a Larch/C Interface Specification Language*. PhD thesis, Concordia University, Department of Computer Science, Montreal, Quebec, Canada, December 1995.

[4] GUTTAG, J., HORNING, J., AND WING, J. The Larch Family of Specification Languages. *IEEE Software 2*, 5 (1985), 24–36.

[5] GUTTAG, J. V., AND HORNING, J. J. Introduction to LCL, A Larch/C Interface Language. Tech. Rep. 74, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, July 1991.

[6] GUTTAG, J. V., AND HORNING, J. J. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[7] HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, 1985.

[8] LEAVENS, G. T. Larch/C++ Reference Manual. Available at ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz., 1995.

[9] MILNER, R. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, New York, NY, 1989.

[10] PERRY, D. *VHDL*, 1st ed. McGraw-Hill, New York, NY, 1991.

[11] TENNENT, R. *Principles of Programming Languages*. Computer Science Series. Prenitce-Hall International, 1981.

APPENDIX P:
Abstract Architecture Representation Using VSPEC*

Phillip Baraona and Perry Alexander
Department of Electrical and Computer Engineering
and Computer Science
The University of Cincinnati
Cincinnati, OH
{pbaraona,alex}@ececs.uc.edu

August 2, 1996

Abstract

Complex digital systems are often decomposed into architectures very early in the design process. Unfortunately, traditional simulation based languages such as VHDL do not allow the impact of these architectural decisions to be evaluated until a complete, simulatable design of the system is available. After a complete design is available, architectural errors are time-consuming and expensive to correct. However, there is an alternative to simulation based techniques: formal analysis of abstract architectures at the requirements level. This paper describes VSPEC's approach for defining and analyzing abstract architectures. VSPEC is a Larch interface language for VHDL that allows a designer to specify the requirements of a VHDL entity using the canonical Larch approach. VHDL structural architectures that instantiate VSPEC entities define abstract architectures. These abstract architectures can be evaluated at the requirements level to determine the impact of architectural decisions. This paper briefly introduces VSPEC, provides a formal definition of VSPEC abstract architectures and presents two examples that illustrate the architectural definition capabilities of the language.

This paper was submitted to the *VLSI Design* journal on February 29, 1996. It was revised and resubmitted on July 25, 1996. Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615-93-C-1316.

1 Introduction

Architectural design decisions made early in a system's design profoundly affect overall design quality. Unfortunately, architecture decisions are rarely evaluated until late in the design process. Simulation-based design languages such as VHDL [5, 12] do not allow evaluation until complete models exist. For large systems, simulatable models appear late in the design process driving up the cost of error correction. These models include not only architectural decisions, but also component design decisions. The ability to analyze architectural decisions as they are made would significantly reduce this cost.

A solution to late architecture evaluation is formal analysis of abstract architectures at the requirements level. An abstract architecture is an interconnected collection of components where the requirements of each component are specified without defining their implementation. Thus, an abstract architecture describes a class of solutions with a common structure rather than a single instance from that class. Formally described abstract architectures can be evaluated early in the design process when architecture decisions are made before component designs exist.

VSPEC [7], a Larch interface language [10] for VHDL [12], is a requirements specification language that includes formal architecture definition support. VSPEC describes the requirements of digital system components using the canonical Larch approach. Each VHDL entity is annotated with a pre- and post-condition to specify the entity's functional requirements. VSPEC-annotated entities can be connected together using a VHDL structural architecture to form abstract architectures. The VHDL architecture indicates interconnection in the traditional manner, but the VSPEC specification defines the requirements of each component instead of a specific design.

The description of a sorting component illustrates the difference between VHDL and VSPEC. In VHDL, the simplest way to describe the function of a sorting component is a behavioral architecture that implements a quicksort, bubble sort or some other sorting algorithm. This is actually a description of "how" the sorting component behaves. In contrast, a VSPEC specification of this

```
entity sort is
  port (input: in element_array;
        output: out element_array);
  includes SortPredicates;
  modifies output;
  sensitive to input'event;
  ensures
    permutation(output'post,input);
    ordered(output'post);
end sort;
```

Figure 1: VSPEC description of a sort entity.

component explicitly describes "what" the device must do without defining "how" it is done. A VSPEC description of a sorting component is shown in Figure 1. It states the output has all the same elements as the input (`permutation(output'post,input)`) and the output is in order (`ordered(output'post)`). Any sorting algorithm may be used to implement these requirements, but VSPEC allows this algorithm to be chosen later in the design process.

Larch interface languages have been developed for a variety of programming languages including C [9], C++ [15] and Modula-3 [14]. At the single component level, VSPEC differs very little from other interface languages. However, defining a Larch interface language for VHDL presents a problem not found in these other languages. In traditional programming languages, a language construct executes after the construct immediately preceding it terminates. In VHDL, there is no implicit execution order among process level constructs and thus no means of determining when a component's pre-condition should hold. VSPEC addresses this problem by allowing a user to define an activation condition in addition to the pre- and post-condition for an entity. When an entity's state satisfies its activation condition, its pre-condition must hold and the entity must perform its specified transformation.

This paper describes VSPEC, concentrating on the language's facilities for describing abstract architectures. Section 2 provides a brief summary of the VSPEC language. Section 3 describes VSPEC abstract architectures, including a definition of the VSPEC state model and a description of how a process algebra (CSP [11]) is used to provide a semantics for the VSPEC activation condition.

Section 4 presents two example VSPEC specifications, concentrating on the architecture representation portions of each specification. Finally, the paper concludes with a discussion of related work and a brief summary.

2 VSPEC

VSPEC is used to describe "what" a digital system should do. It adds a requirements definition capability to VHDL entities analogous to the requirements definition capability that Larch interface languages add to traditional procedure and function signatures. As shown in Figure 2, the requirements of a VHDL entity can be defined by describing a relationship from the current inputs and state of the system to the outputs and the next state. This section describes how $F(x, s)$ and s are defined in VSPEC and contrasts these definitions with VHDL definitions of $F(x, s)$ and s.

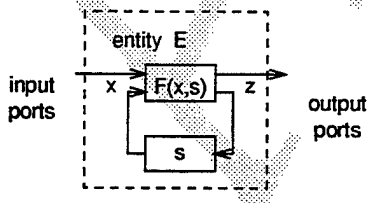


Figure 2: State-based specification model.

As shown in the `find` entity of Figure 3, a VHDL entity defines an interface. The output of `find` should be the element from the `input` array with the same key as the `key` input. A VHDL `entity` does not describe functional information such as this. The `entity` only defines the component's interface.

```
entity find is port
  (input: in element_array;
   key: in keytype;
   output: out element);
end find;
```

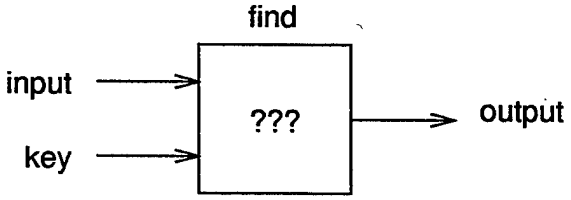


Figure 3: A VHDL `entity` defining the interface for a `find` component.

```
architecture behavior of find is
begin
  process (input,k)
  begin
    for i in input'range loop
      if key = input(i).key then
        output <= input(i);
        exit;
      end if;
    end loop;
  end process;
end behavior;
```

Figure 4: A behavioral VHDL `architecture` defining the `find` component's behavior.

The VHDL `architecture` construct describes the function of a component by associating behavior and/or structure with an `entity`. Figure 4 is a behavioral VHDL description of the `find` component's function. In terms of the state model in Figure 2, this architecture describes $F(x, s)$ as a linear search algorithm. This looks very similar to a C or Pascal function describing "how" the system behaves. Unfortunately, this operational description biases the system towards a particular implementation. Since VSPEC's purpose is requirements specification, it is undesirable to bias the system to a particular implementation this early in the design process.

VSPEC eliminates this problem by allowing a user to declaratively specify the requirements of a digital system. Seven clauses annotate the VHDL entity construct to allow the specification of "what" a component should do instead of VHDL's description of "how" the component performs this function. The `requires`, `ensures` and `sensitive to` clauses are used to specify the device's functional requirements. Non-functional constraints are described in the `constrained by` and `modifies` clauses. The component's internal state is declared in the `state` clause and the `includes` clause is used to make types and operators from a Larch shared language description visible in a VSPEC component. The remainder of this section briefly summarizes these clauses. For a more complete description of the VSPEC clauses, see one of the other VSPEC references. [1, 7]

Component function is described in the `requires` and `ensures` clauses. The `requires` clause defines a pre-condition over inputs and state variables while the `ensures` clause defines a post-

```
entity find is port
  (input: in element_array;
   k: in keytype;
   output: out element);
  includes Element(element,keytype,
                   element_array);
  modifies output;
  requires true;
  ensures forall (e : element)
    (output = e implies
       (e.key = k
       and elem_of(e,input)));
  constrained by
    power <= 5 mW
    and k<->output <= 5 Ms
    and heat <= 10 mW
    and clock <= 50 MHz;
end search;
```

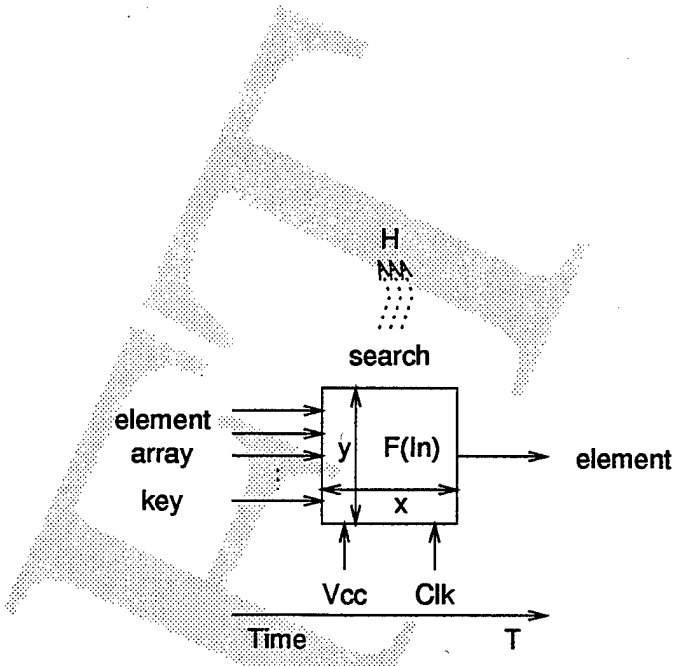


Figure 5: The `find` entity annotated with a VSPEC definition.

condition over inputs, outputs and state variables. The **ensures** clause defines legal outputs and the next state when the **requires** clause is satisfied. A component's user is responsible for making certain the **requires** clause is satisfied whenever the component is in use. When the **requires** clause is satisfied, the described **entity** is responsible for making the **ensures** clause true.

Let σ be the state of a VSPEC entity as defined by its ports and state variables. If $I(\sigma)$ is the **requires** predicate and $O(\sigma,\sigma')$ is the **ensures** predicate, then the VSPEC annotation defines the following requirements:

$$\forall\,\sigma \cdot \exists\,\sigma' \cdot I(\sigma) \Rightarrow O(\sigma,\sigma') \tag{1}$$

$F(\sigma)$ is an implementation of these requirements if the following condition holds:

$$\forall\,\sigma \cdot I(\sigma) \Rightarrow O(\sigma,F(\sigma)) \tag{2}$$

A VSPEC description of a `find` component is shown in Figure 5. Notice that the **requires** clause predicate is **true** meaning this entity will function correctly for any set of inputs of the proper type. The **ensures** clause predicate states that the output element has the same key as the **k** input and **output** is in the input sequence. In terms of the state model in Figure 2, this defines the requirements of $F(x,s)$, but unlike the VHDL description, it does not describe how to

implement the component.

The VSPEC `sensitive to` clause[1] is used to define when a component in an abstract architecture is active. When the `sensitive to` clause predicate is true, a component's pre-condition must hold and an implementation must satisfy the post-condition. A more precise description of this clause can be found in Section 3.

Performance constraints are described in the `constrained by` and `modifies` clauses. Constraints define requirements such as clock speed or layout area that are not part of the functional description. The `constrained by` clause defines relations over constraint variables. Currently, the defined constraint variables include power consumption, clock speed, area, pin-to-pin timing, and heat dissipation. Constraint theories written in the Larch Shared Language (LSL) [10] define each constraint type. Users may define their own constraints and theories if desired. The `modifies` clause lists variables, ports and signals whose values may be changed by the `entity`. This clause is useful when specifying whether an entity modifies a shared variable. The list of objects an entity modifies is not a traditional performance constraint, but this does restrict the set of potential solutions. Examples of the `constrained by` and `modifies` clauses are shown in Figure 5.

The state of a VSPEC entity is described by the port definition and variables in the `state` clause. In VHDL, ports maintain their values between `entity` invocations. Thus, port values from the previous state may be accessed in the current state. The `state` clause is used to define internal state variables that are used in the VSPEC definition only. These variables maintain state information that is not recorded in port values. When a VSPEC specification is refined into a VHDL architecture, these internal state variables will be refined into signals or variables that represent the same information. The `state` clause variable declaration represents this information during the requirements specification phase of the entity's design. An example of the state clause can be found in the Move Machine description in Section 4.2.

[1] Previous versions of VSPEC [1, 2, 7] did not have a `sensitive to` clause.

The includes clause is the final VSPEC clause.[2] This clause is used to include LSL definitions in a VSPEC description or VHDL package declaration (see Section 4.2.)[3] LSL is used to define the types and functions used in a VSPEC specification. An example of the includes clause is shown in Figure 5 and its syntax is the keyword includes followed by a list of trait references. The syntax of a trait reference is similar to a trait reference in LSL. It consists of the trait name followed by an optional parameter list. The parameter list is used to rename LSL names to a name visible in the VSPEC entity. Thus, an integer stack is included in a VSPEC specification with this includes clause: `includes Stack(integer, int_stack)`.

3 Architectures

The previous section briefly described how VHDL and VSPEC are used to define the requirements of a single device in a digital system. The behavior of a device can also be described by decomposing it into smaller pieces and connecting these pieces together to form an architectural description of the device. This architectural description represents a refinement of the device's behavioral VHDL/VSPEC description. VHDL provides convenient facilities for defining architectural descriptions. This section briefly discusses these facilities and then describes how VSPEC uses them to form an abstract architecture.

3.1 VHDL Structural Architectures

VHDL uses structural architectures to represent component composition. A structural architecture describes how sub-components are connected together to form a larger component. Figure 6 shows a structural architecture for `find`. Unlike the behavioral representation in Figure 4, this architecture indicates that a `sort` component connected to a `search` component implements the `find` function.

[2] Previous versions of VSPEC [1, 2, 7] also contained a based on clause. The modified syntax of the includes clause described here made the based on clause obsolete.

[3] Allowing includes clauses in package declarations is a change from previous versions of VSPEC. [1, 2, 7]

This structural architecture should perform the same function as that specified in the behavioral description.

The VHDL `component` construct defines each component used in a structural architecture. The `structure` architecture of `find` in Figure 6 declares two types of components that are used in this architecture: `sorter` and `searcher`. One instance of each of these components (named `b1` and `b2`) is created in the body of this architecture. The port maps of these component instances are used to indicate how the components are connected together. In the `structure` architecture for find, the system's input array is connected to the `sorter` input and the `sorter` output is connected to internal architecture signal `y`. The signal `y` and system input `k` are inputs to the `searcher` component. The output of the `searcher` is connected to the device output.

The VHDL `configuration` construct is used to bind entity-architecture pairs to component instances. In this example, the `test_struct` configuration binds the bubble sort defined by entity `sort` with architecture `behavior` to the `b1` instance of the `sorter` component. Similarly, the binary search defined by entity `bin_search` with architecture `behavior` is bound to the `b2` instance of `searcher`. If there were other architectures for these two entities (such as a structural architecture), a different configuration could have been specified stating that the components in `structure` mapped to these architectures. Entirely different entities could even have been defined.

Since a structural architecture only defines dataflow between components, an additional mechanism must be provided to define when a component activates. VHDL accomplishes this with sensitivity lists and `wait` statements. A sensitivity list contains a list of signals. Whenever an event occurs on one of these signals, the process resumes execution. The `behavior` architecture for `sort` is sensitive to its single input, while `bin_search` is sensitive to its input array and key value. This means the sort component sorts its input only when new input arrives. Likewise, a search occurs only when the key value or input array changes. A `wait` statement achieves the same result by waiting on signal conditions or for a specific simulation time interval. In this example, `wait` statements could replace sensitivity lists by removing the sensitivity lists and placing `wait`

```
architecture structure of find is
  component sorter
    port (input: in element_array;
          output: out element_array);
  end component;
  component searcher
    port (input: in element_array;
          key: in keytype;
          value: out element);
  end component;
  signal y: element_array;
begin
  b1: sorter port map(input,y);
  b2: searcher port map(y,k,output);
end structure;

entity sort is
  port (input: in element_array;
        output: out element_array);
end sort;

architecture behavior of sort is
begin
  process(input) begin
    -- Behavioral VHDL description
    -- of a bubble sort
  end process;
end behavior;
```

```
entity bin_search is
  port (input: in element_array;
        key: in keytype;
        value: out element);
end bin_search;

architecture behavior of bin_search is
begin
  process (input,key) begin
    -- Binary search algorithm
    -- definition in behavioral VHDL
  end process;
end behavior;

configuration test_struct of find is
  for structure
    for b1:sorter use entity
      work.sort(behavior);
    end for;
    for b2:searcher use entity
      work.bin_search(behavior);
    end for;
  end for;
end test_struct;
```

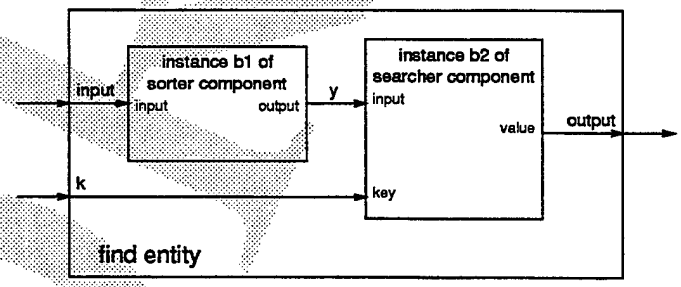


Figure 6: A VHDL architecture representing the composition of a sorting component and a binary search component implementing the find function.

statements referencing the same signals at the end of the process definitions.

These constructs allow VHDL to support architecture representation. Component declarations describe the inputs and outputs of each component type used in the architecture. Instances of these components are created in the architecture body and configurations are used to map component instances to an entity/architecture pair. Net lists indicate signal flow between component instances while sensitivity lists or `wait` statements synchronize component actions.

3.2 VSPEC Abstract Architectures

VHDL structural architectures containing VSPEC annotated components specify abstract architectures. The VHDL architecture remains unchanged indicating component instantiation and connections. However, a VHDL architecture is not assigned to each component instance in the architecture. Instead, the configuration defines that each component references an entity with an architecture called `VSPEC`. This signifies that at the current point in the design, the requirements of this component are known (via the VSPEC description) but no implementation has been defined. [4]

The `structure` architecture of `find` shown in Figure 6 becomes an abstract architecture by referencing VSPEC definitions of the instantiated components. Figure 7 shows VSPEC `entity` definitions for the `sort` and `bin_search` components in Figure 6. A new configuration, `test_vspec`, has been defined for the `find` entity. It specifies that the VSPEC descriptions of `sort` and `bin_search` should be used instead of a specific architecture for these two entities. This configuration describes an abstract architecture for the `find` component. Any implementation satisfying the VSPEC requirements of `sort` and `bin_search` may be associated with the `entity` definitions. The architectures specified in Figure 6 represent one such solution, but there are many others.

The VSPEC description of sort specifies the requirements for a sorting component: the input and output must have all the same elements (i.e. output is a permutation of input) and the output must

[4] This is different than leaving the entity open. When a VHDL entity is left open, the design is being deferred. At the current point in the design, nothing is known about the function of the entity. In contrast, the requirements of a VSPEC entity are known, even though an implementation is not.

be in order. In a similar fashion, the bin_search specification states that whenever the component input is sorted, the component must ensure that the output element contains the same key as the k input and this element is an element of the input array. The requires and ensures clauses of these entities use two predicates (permutation and ordered) to define these requirements. These predicates are defined in the LSL trait SortPredicates which is included in both VSPEC entities.

```
entity sort is
  port (input: in element_array;
        output: out element_array);
  includes SortPredicates;
  modifies output;
  sensitive to input'event;
  ensures
    permutation(output'post,input);
    ordered(output'post);
end sort;

entity bin_search is
  port (input: buffer element_array;
        key: in keytype;
        output: out element);
  includes SortPredicates;
  modifies value;
  sensitive to k'event or input'event;
  requires ordered(input);
  ensures output = e iff (e.key=k and
                element_of(e,input));
end bin_search;

configuration test_vspec of find is
  for structure
    for b1:sorter use entity
      work.sort(VSPEC);
    end for;
    for b2:searcher use entity
      work.bin_search(VSPEC);
    end for;
  end for;
end test_struct;
```

Figure 7: VSPEC definitions for the sort and bin_search components in the find architecture.

Although a VHDL architecture referencing VSPEC definitions defines components and interconnections, additional information must be added to specify when the VSPEC components activate. In traditional sequential programming, a language construct "executes" following termination of the construct preceding it. For correct execution, a construct's pre-condition must be satisfied when the preceding construct terminates. In hardware systems, components exist simultaneously and behave as independent processes. No predefined execution order exists, thus there is no means for determining when a component's pre-condition should hold. Consider the find example. The pre-condition of bin_search need hold only when sort has completed its transformation. At all other times, bin_search need only maintain its state.

VHDL provides sensitivity lists and wait statements to synchronize entity execution. VSPEC

achieves the same end using the `sensitive to` clause. The `sensitive to` clause contains a predicate called the activation condition indicating when an entity should begin executing. Effectively, the activation condition defines when a VSPEC annotated `entity`'s pre-condition must hold. When the `sensitive to` predicate is true, the pre-condition must hold and the implementation must satisfy the post-condition. When the `sensitive to` predicate is false, the entity makes no contribution to the next state of the system. Like the `requires` and `ensures` clauses, the `sensitive to` predicate is defined over entity port definitions and variables defined in the `state` clause.

Recall that the structural VHDL architecture for `find` (Figure 6) specified that the `sort` component should only activate when its input changes and the binary search component activates when one of its inputs changes. Without the `sensitive to` clause, specifying this behavior in VSPEC would not be possible. Note the `sensitive to` clauses defined in the VSPEC description of `find` in Figure 7. In VSPEC, a signal's `'event` attribute is true if the signal changed value from the previous state. Thus, both components activate whenever any of their inputs change value.

3.3 Architecture Model Semantics

The previous section provided an informal description of how VSPEC can be used to define an abstract architecture. This section provides a more precise, formal definition of the concepts presented above. First, the state of a VSPEC description is defined. After this, a precise definition of how the `sensitive to`, `requires` and `ensures` clauses define a transformation over this state is presented. The section concludes with a simple example that illustrates these points.

3.3.1 State Definition

The state definition for an entity is a map from port, signal and variable names to their values. There are three different views of an entity state: (1) abstract; (2) component; and (3) concrete state. The abstract state is defined by a VSPEC description of an entity. The component state is the state of a single component in an abstract architecture and the concrete state represents the

state of all components of an abstract architecture.

The abstract state includes the ports and state variables of an entity. The VSPEC **sensitive to**, **requires** and **ensures** clause predicates are defined over elements of the abstract state of the entity. The component state applies to an entity included as a component in a structural architecture. The component state is formed by taking the entity's abstract state and subjecting it to the renaming imposed by the signals the component is connected to in the architecture. This component state is used to construct the concrete state of the structural architecture. The concrete state is the union of the component states for all of the components in an architecture. This structural architecture represents a refinement of the VSPEC definition of the entity. There is an abstraction function mapping the concrete state of the structural architecture to the abstract state defined by the VSPEC description of the entity the structural architecture refines.

Consider the VSPEC entity in Figure 8. The abstract state of the three entities in this figure are the inputs, outputs and state variables of the entities. Thus, the abstract states of these entities are:

$$
\begin{aligned}
ABSTRACT_{system} &= \{sys_in \mapsto i_0, sys_out \mapsto i_1, sys_state \mapsto i_2\} \\
ABSTRACT_{comp1} &= \{in1 \mapsto i_3, result \mapsto i_4, c1_state \mapsto i_5\} \\
ABSTRACT_{comp2} &= \{in1 \mapsto i_6, in2 \mapsto i_7, result \mapsto i_8, c2_state \mapsto i_9\}
\end{aligned}
$$

where $i_0, i_1, ...i_9$ are all integers. As shown, the state is a map from names to values. However, for the purpose of clarity we will show just the names that form the various states throughout the rest of this paper.

Within the **struct** architecture for the **system** entity, the **A**'s component state (the first instance of **comp1**) is found by taking **comp1**'s abstract state and performing the renaming defined by the signals the component is connected to. In this case, **in1** is connected to **sys_in** and **result** is connected to signal **x**. Thus, in the context of the **struct** architecture, **in1** of component instance **A** should be replaced by **sys_in** and **result** replaced by **x**. A similar renaming can easily be found for the inputs and outputs of the other components in the **struct** architecture. The renaming for

```
entity system is
  port (sys_in : in integer;
        sys_out : out integer;);
  state (sys_state : integer;);
end system;

entity comp1 is
  port(in1 : in integer;
       result : out integer;);
  state (c1_state : integer;);
end comp1;

entity comp2 is
  port(in1, in2 : in integer;
       result : out integer;);

  state (c2_state : integer;);
end comp2;
```

```
architecture struct of system is
  component comp1
  port (in1 : in integer;
        result : out integer;);
  end component;

  component comp2
  port (in1, in2 : in integer;
        result : out integer;);
  end component;

  signal x, y : integer;

begin
  A : comp1 port map(sys_in,x);
  B : comp1 port map(x,y);
  C : comp2 port map(x,y,sys_out);
end struct;
```

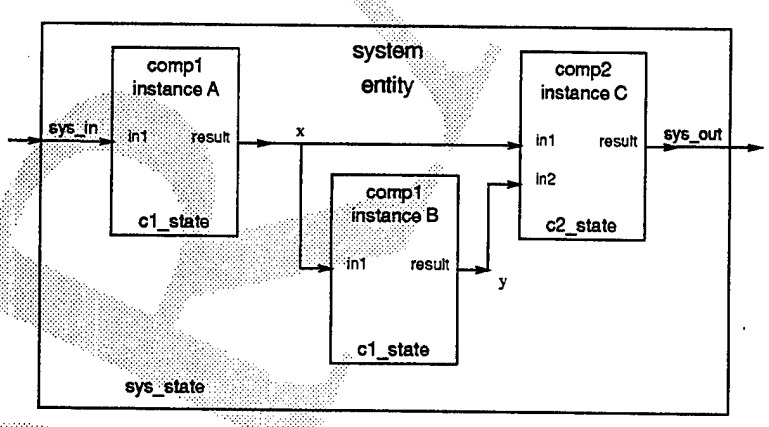


Figure 8: Example VSPEC entity used to explain the differences between abstract, component and concrete state.

the other components is shown in the definition A and B's component states below.

Since the struct architecture has more than one instance of the comp1 entity, the state variables of comp1 must be renamed to form the component state. This renaming avoids conflicts when forming the concrete state of the struct architecture. To simplify matters, we will always rename a component's state variables even if there is only one instance of an entity in the architecture. A number of renaming functions could be chosen, but the one used here is the state variable name in the abstract state subscripted with the instance label from the architecture. The component states of the components in the struct architecture are:

$$
\begin{aligned}
COMPONENT_A &= ABSTRACT_{comp1}[in1/sys_in, result/x, c1_state/c1_state_A] \\
&= \{sys_in, x, c1_state_A\} \\
COMPONENT_B &= ABSTRACT_{comp1}[in1/x, result/y, c1_state/c1_state_B] \\
&= \{x, y, c1_state_B\} \\
COMPONENT_C &= ABSTRACT_{comp2}[in1/x, in2/y, result/sys_out, c2_state/c2_state_C] \\
&= \{x, y, sys_out, c2_state_C\}
\end{aligned}
$$

We are now ready to form the concrete state of the struct architecture for the system entity. The concrete state is simply the union of the component states for each component in the architecture:

$$
\begin{aligned}
CONCRETE_{struct_{system}} &= COMPONENT_A \cup COMPONENT_B \cup COMPONENT_C \\
&= \{sys_in, sys_out, x, y, c1_state_A, c1_state_B, c2_state_C\}
\end{aligned}
$$

Since an abstract architecture represents a refinement of the requirements specified by VSPEC, an abstraction function can be defined to map the concrete state of the architecture the abstract state defined by the VSPEC description.

Together, the abstract, component and concrete states represent the state of a VSPEC component. The examples in Sections 3.3.3 and 4 use these definitions to describe how a VSPEC description behaves.

3.3.2 Transform Definition

The transform performed by a VSPEC architecture is defined by the `sensitive to`, `requires` and `ensures` clauses. The formal definition of the `requires` and `ensures` clauses was discussed in Section 2. It is very similar to the transform defined by a traditional Larch interface language. As described in Section 3.2, the `sensitive to` clause is used to synchronize components and define when the `requires` clause predicate must be satisfied.

Formally, synchronization is easily represented using a traditional process algebra such as CSP [11]. Events are defined as changes in the state of the entity. Assume that $F(St)$ is a function between two states of entity P that implements the requirements specified in P's `requires` and `ensures` clauses (i.e. $F(St)$ satisfies Equation 2). The process defined by entity P with a `sensitive to` predicate of $S(St)$ in any state St is:

$$P_{St} = t : SEN \rightarrow P_{F(St)} \tag{3}$$

where SEN is the set of states that satisfy P's `sensitive to` clause: $SEN = \{t|S(t)\}$. Thus, a process in state St first waits for its `sensitive to` clause to be satisfied and then behaves like the same process in the state defined by applying F to the current state.

Equation 3 defines a CSP process that describes the behavior of a single VSPEC entity. CSP's concurrency operator ($\|$) is used to define a process that describes the behavior of an architecture of VSPEC components. Let $P_0, P_1, ..., P_n$ be the processes represented by Equation 3 for the set of VSPEC component instances in architecture $\mathcal{P}$. The process that represents architecture $\mathcal{P}$ is:

$$\mathcal{P} = P_0 \parallel P_1 \parallel ... \parallel P_n \tag{4}$$

Thus, each component in the architecture executes in parallel. Since a component activates only when its `sensitive to` clause predicate is true, this predicate is used to synchronize component execution.

```
entity example is
  port(i: in integer; o: out integer);
end example;

architecture structural of example is
  component c1
    port(input: in integer;
         output: out integer);
  end component;
  component c2
    port(input: in integer;
    output: out integer);
  end component;
begin
  b1: c1 port map(i,y);
  b2: c2 port map(y,o);
end structural;
```

```
entity c1 is
  port (x: in integer; z: out integer);
  modifies z;
  sensitive to x'event;
  requires I1(x);
  ensures O1(x, z'post);
end c1;

entity c2 is
  port (x: in integer; z: out integer);
  modifies z;
  sensitive to x'event;
  requires I2(x);
  ensures O2(x, z'post);
end c2;
```

Figure 9: Specification of two components connected serially.

3.3.3 Formal Model Example

This section presents a simple example to explain how the concrete state of a VSPEC architecture changes as its inputs are modified by external components. Consider the architecture shown in Figure 9. The abstract, component and concrete state of the elements of this architecture are:

$$\begin{aligned} ABSTRACT_{c1} &= \{x, z\} \\ ABSTRACT_{c2} &= \{x, z\} \\ \\ COMPONENT_{c1} &= \{i, y\} \\ COMPONENT_{c2} &= \{y, o\} \\ \\ CONCRETE_{structural_{example}} &= \{i, o, y\} \end{aligned}$$

The transformation performed by an architecture is defined from the components comprising it. Formally, the component requirements for c1 and c2 are defined as:

$$\forall x : integer, \exists z : integer \cdot I_1(x) \Rightarrow O_1(x, z'post)$$
$$\forall x : integer, \exists z : integer \cdot I_2(x) \Rightarrow O_2(x, z'post)$$

The renaming defined by the architecture that is used to create the component state from the abstract state of an architecture can also be applied to these two equations. In this example, this

defines the following logical requirements for c1 and c2:

$$\forall\, i : integer, \exists\, y : integer \cdot I_1(i) \Rightarrow O_1(i, y'post)$$
$$\forall\, y : integer, \exists\, o : integer \cdot I_2(y) \Rightarrow O_2(y, o'post)$$

The renaming function is also applied to the `modifies`, `state` and `sensitive to` clause of c1 and c2. After this renaming, the logical definitions of each component are expressed in the same name space as the concrete state of the system.

Assume that a, b and c are integer constants and that $f(x)$ and $g(x)$ are functions that satisfy requirements for c1 and c2 respectively. Let the initial concrete state of the system be $\mathcal{S}_0 = \{i \mapsto a, y \mapsto b, o \mapsto c\}$ and let $i'event$ be true and $y'event$ be false. This means that c1's `sensitive to` clause is satisfied and c1's pre-condition must hold. c1 will then make its post-condition hold in the next state. Instantiating the requirements for c1 gives:

$$\exists\, z : integer \cdot I_1(a) \Rightarrow O_1(a, z) \tag{5}$$

Knowing that $f(x)$ satisfies c1's requirements and assuming $I_1(a)$ is true implies that $O_1(a, f(a))$ is also true. Additionally, $y'event$ is known to be false so c2 maintains its state and o does not change in the next state. Thus, one potential next state for this system is $\mathcal{S}_1 = \{i \mapsto a, y \mapsto f(a), o \mapsto c\}$. Because the function f is one of potentially many functions satisfying c1, we cannot claim that this is the only possible next state.

Since y changed values from $\mathcal{S}_0$ to $\mathcal{S}_1$, the predicate $y'event$ is true in $\mathcal{S}_1$. Additionally, i did not change values in $\mathcal{S}_1$ implying that $i'event$ is false in $\mathcal{S}_1$. Thus, only component c2 activates in state $\mathcal{S}_1$.

Using the same reasoning used for $\mathcal{S}_1$, values for $\mathcal{S}_2$ can be produced. Assuming that $f(a)$ satisfies $I_2(f(a))$ and knowing $g(x)$ satisfies c2's requirements makes $O_2(f(a), g(f(a)))$ true. The input value i has not changed, c1 maintains its state implying y does not change, and $g(f(a))$ satisfies c2's output condition. Thus, $\mathcal{S}_2 = \{i \mapsto a, y \mapsto f(a), o \mapsto g(f(a))\}$ is a potential next state for the system.

An interesting exercise is defining what happens when the input value i changes between states S_0 andS_1. Assume that i changes value from a to d making $S_1 = \{i \mapsto d, y \mapsto f(a), o \mapsto c\}$. Now *i'event* is true in S_1 and both components execute on values from S_1. In this case, $S_2 = \{i \mapsto d, y \mapsto f(b), o \mapsto g(f(a))\}$. Note the value of o does not change from the previous example because the next state is defined only on variables defined in the current state. Using this model eliminates difficulty caused by instantaneous feedback and "pipelined" update functions. VHDL solves this same problem by allowing an infinite number of delta delays between major clock cycles of the simulation.

3.4 Generating Proof Obligations

The VSPEC formal model can be used to verify that a system's abstract architecture description satisfies the requirements described by the VSPEC specification of the system. This verification provides evidence that the abstract architecture description satisfies the abstract VSPEC specification. Finding such evidence depends on: (1) having the system requirements I and O; and (2) relating a concrete state produced by the abstract architecture with the abstract state specified for the system. A system's VSPEC description provides I and O. The abstraction function from the concrete to the abstract state provides the means for comparing the abstract and concrete states.

Weak bisimulation [19] is used as the correctness criteria when attempting to verify that an abstract architecture satisfies a VSPEC description. As shown in Figure 10, weak bisimulation requires that some sequence of state changes in the concrete state of the system result in the correct single state change in the abstract state. Only the first and last of the concrete states are significant. The system may pass through any concrete state as long as the abstraction function applied to the final concrete state results in the correct abstract state as defined by the abstract specification.

In CSP, the sequence of states a VSPEC entity passes through is called a trace. A CSP trace of process P is a finite sequence of symbols representing the events processed by P. VSPEC events

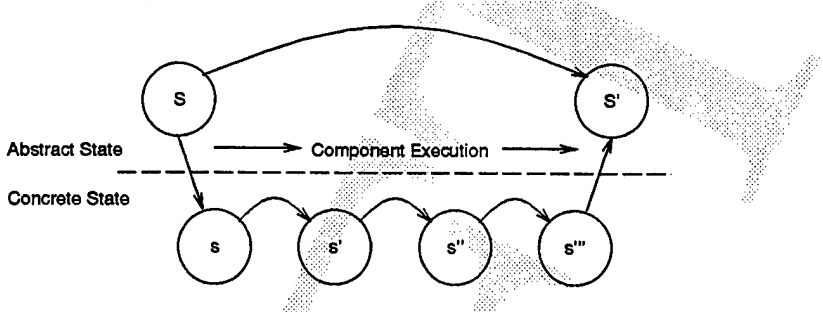


Figure 10: Concrete state changes associated with a single abstract state change.

are changes in state and they are represented in a trace by the state the entity changes to. Thus, a VSPEC entity satisfies the weak bisimulation criteria if two conditions hold for all traces of the abstract architecture. The first condition is that the abstraction function applied to the initial element of each trace must result in an abstract state that satisfies the abstract pre-condition. The second condition is that the final element of each trace must either have an abstract projection that satisfies the abstract post-condition or there must be some legal sequence of states that can be appended to the trace to form another trace. This ensures that the concrete state eventually reaches a state where the abstract specification is satisfied.

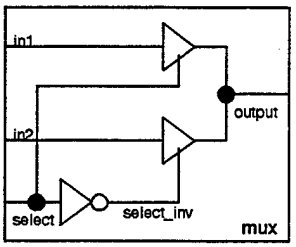
4 Examples

This section presents two examples that illustrate how VSPEC can be used to describe an abstract architecture. The first example is a simple tri-state buffer description that is used to define a simple 2 input multiplexor. This example illustrates what happens when multiple sources drive a single value in a VSPEC abstract architecture. The second example is the description of a simple CPU called the Move Machine. This example illustrates shows a VSPEC description that is decomposed into an abstract architecture.

```
entity buffer is
  port (input: in integer;
        control: in boolean;
        output: out integer);
  sensitive to control'event or input'event;
  ensures control implies output'post = input;
end buffer;
```

Figure 11: VSPEC description of a simple buffer.

```
entity mux is
  port (in1, in2: in integer;
        select: in boolean;
        output: out integer);
  sensitive to in1'event or
    in2'event or select'event;
  ensures
    (select and output'post = in1) or
    (not select and output'post = in2);
end mux;
```




```
architecture struct of mux is
  component buffer
  port (input: in integer;
        control: in boolean;
        output: out integer);
  end component;
  component not
  port (input: in boolean;
        output: out boolean);
  end component;
  signal select_inv : boolean;
begin
  b1: buffer
    port map(in1,select,output);
  b2: buffer
    port map(in2,select_inv,output);
  n1: not
    port map(select,select_inv);
end struct;
```

Figure 12: VSPEC and abstract architecture description of a 2-input mux.

4.1 Buffer and Multiplexor Example

A VSPEC description of a simple buffer is shown in Figure 11. In this example, input and output are both integers, but the specification could also be used if input and output were of any other type. When control is true, this device passes input to output. When control is false, the device places no requirements on the value of output in the next state. The specification allows for output to maintain its current value in the next state, but the specification also allows an external device to change the value of output. Consider using this buffer as a component in the abstract architecture description of the multiplexor in Figure 12.

This figure shows both a VSPEC description of a multiplexor as well as a refinement of this description into an abstract architecture. The VSPEC entity mux is a straightforward description of

a multiplexor. The `struct` architecture uses two instances of `buffer` and a `not` gate to decompose the multiplexor into an abstract architecture.

Careful examination of this description reveals a very subtle but important point about VSPEC specifications and multiply driven signals. If a component description does not restrict the value of an output signal in the next state, other components in the system can still change the value of this signal without violating the component description. Suppose that the concrete state of the architecture is:

$$CONCRETE_{struct_{mux}} = \{in1 \mapsto 7, in2 \mapsto 3, select \mapsto true, output \mapsto 7, select_inv \mapsto false\}$$

so that the abstract state of `buffer` instance `b1` is:

$$ABSTRACT_{b1} = \{input \mapsto 7, control \mapsto true, output \mapsto 7\}$$

Assume that some external device changes the `select` input to false. This causes `buffer` instance `b1`'s `control` input to change to false which activates the buffer. This device must now make its `ensures` clause true in the next state. Since `control` is false, the `ensures` clause will be true in the next state for any value of `output`. Thus, `buffer` instance `b2` can change the `output` signal of the architecture to 3 without violating `b1`'s specification. The next state of the device is:

$$CONCRETE_{struct_{mux}} = \{in1 \mapsto 7, in2 \mapsto 3, select \mapsto false, output \mapsto 3, select_inv \mapsto true\}$$

Thus, the `output` signal has changed values even though the `b1` buffer instance does not cause it to do so. Even though `b1` does not force a change in state, it does not prohibit one either. An external device (buffer instance `b2`) has caused the `output` signal to change values. The specification of `b1` allows this change to occur.

This description may not seem correct to an experienced VHDL user because the output signal is driven by two sources, but no resolution function is specified. Although this is illegal in VHDL, it is allowed in VSPEC. In most cases, the CSP statement that defines a VSPEC entity's contribution to the next state of the system will define a single value for every signal, but a VSPEC description may allow more than one value for a specific signal. This is legal VSPEC because VSPEC is a specification language, not a simulation language like VHDL. This implies that a VSPEC specification does

not need to deterministically define a single value for every signal in the system. It is certainly possible to do this with VSPEC by defining the requirements of resolution functions, but a VSPEC specification could allow a signal to be driven to two (or more) different values. In these cases, a designer implementing the specification may chose to drive the signal to any of its allowed values.

4.2 The Move Machine

A more complex example is the specification of a Move Machine [22]. The Move Machine is a simple CPU that moves data from one memory location to another. It uses four instructions: jump, load register from memory, store register to memory, and halt and four addressing modes: absolute, immediate, indirect and relative. Although the Move Machine is a simple device, its structure reflects how a more complex system might be represented.

The first step in specifying the Move Machine is representing it as a simple instruction interpreter (Figure 13). At this level, only one VSPEC annotated `entity` describes the execution of each instruction and addressing mode. This entity contains state variables to store the current register contents and the value of the instruction pointer. The `sensitive to` clause states that the machine activates when its `start` or `reset` input is on or when the value of the instruction pointer changes. The rather complex `ensures` clause predicate defines how the machine behaves for each instruction and addressing mode. An external entity would use this component by first applying the `reset` signal and then the `start` signal. This causes the machine to begin executing the instruction in memory location 0. The result of each instruction (except `halt`) cause the contents of the instruction pointer to change which activates the machine again in the next state. This continues until a `halt` instruction is processed, causing the machine to stop.

One thing to note about this specification is the `use` clause on the first line. In VHDL, types and functions can be declared in separate packages. These packages are then included in entity and architecture descriptions with the `use` clause. The `mm_types` package referenced in this example is shown in Figure 14. An interesting aspect of this package is the use of incomplete types to specify

```
use work.mm_types.all;
entity mm is
 port (reset,start : in boolean;
       mem: inout memory);
 state (ip : address;
        reg : regfile);
 sensitive to start or reset or
   ip'event;
 ensures

 (reset and ip'post = 0) or

 (not reset and

  ((ins(mem(ip)) = jump and
    ip'post=addr(mem(ip)))

  or (ins(mem(ip)) = load and
  ((am(mem(ip)) = ab and
    reg(rnum(mem(ip)))'post =
     addr(mem(ip))) or
   (am(mem(ip)) = imm and
    reg(rnum(mem(ip)))'post =
     mem(ip +1)) or
   (am(mem(ip)) = ind and
    reg(rnum(mem(ip)))'post =
     mem(addr(mem(ip)))) or
   (am(mem(ip)) = rel and
    reg(rnum(mem(ip)))'post =
     mem(ip + addr(mem(ip))))))
  or (ins(mem(ip)) = store and
  ((am(mem(ip)) = ab and
    mem(addr(mem(ip)))'post =
     reg(rnum(mem(ip)))) or
   (am(mem(ip)) = imm and
    mem(ip +1) =
     reg(rnum(mem(ip)))) or
   (am(mem(ip)) = ind and
    mem(mem(addr(mem(ip)))) =
     reg(rnum(mem(ip)))) or
   (am(mem(ip)) = rel and
    mem(ip + addr(mem(ip))) =
     reg(rnum(mem(ip))))))
 and ((ins(mem(ip)) = store or
   ins(mem(ip)) = load) and
   ((am(mem(ip)) /= imm and
     ip'post = ip'post+1)
    or (am(mem(ip)) = imm and
     ip'post = ip'post+2))));
end mm;
```

Figure 13: The Move Machine requirements represented as an instruction interpreter.

```
package mm_types is
  type address;
  type word;
  includes Instruction(word,address,integer);
  type control is (fetch,decode,execute,halt);
  type memory is array(0 to 256) of word;
  type regfile is array(0 to 15) of word;
end mm_types;
```

Figure 14: Package declaring types used in the Move Machine.

address and word. VHDL uses incomplete types to allow references to a type before the type is completely defined (such as in an access type). One use of this is to allow a record to contain a pointer to another record of the same type (i.e. to construct a list).

In VSPEC, incomplete types are used for a slightly different purpose. The type definitions for address and word are incomplete because no implementation is defined. They are declared to be types, but no additional information is provided. These incomplete types will be given characteristics by the specification, but no specific implementation is implied or mandated. Thus, the designer must select an implementation at a lower abstraction level. Using incomplete types allows the designer to specify a type's characteristics without specifying its implementation.

The characteristics of the address and word types are defined in the LSL Instruction trait. This trait is included in mm_types using a VSPEC includes clause (see Section 2) and the trait is shown in Figure 15. The Instruction trait provides definitions for conversion functions that allow instructions, register numbers and addresses to be obtained from memory words. In the final format of the Move Machine instructions (not shown in this paper), this would be implemented by defining which bits of a memory word encode the instruction, register number and address. However, when specifying the initial requirements of the device, such details should not be considered. All that must be specified is that instructions, register numbers and addresses can be obtained from memory words. This is exactly what the LSL description allows us to say.

Once the Move Machine's initial requirements are defined, the device can be broken up into an abstract architecture and each of the components can be synthesized individually. For a CPU such

```
Instruction(W,A,N): trait

includes
  Natural(N)
  mode enumeration of abs, imm, ind, rel
  instruction enumeration of halt, jump, load, store

introduces
  am: W → mode
  addr: W → A
  ins: W → instruction
  rnum: W → N
```

Figure 15: LSL support functions for treating memory contents as instructions. Basic types and conversions are defined.

as the Move Machine, one such architecture is the canonical fetch–decode–execute structure. An instruction is retrieved, the addressing modes are decoded and dereferenced, and the instruction is executed on its operands. Effectively, the Move Machine is now three components that execute in sequence.

Figure 16 shows the fetch–decode–execute architecture for the Move Machine. The signals `mem`, `reg`, `IP`, `IR`, `EA` and `CNTL` exchange memory, registers and control values between components. The `requires` and `ensures` clauses for each component describe transformations performed on memory and register values while the `sensitive to` clauses uses the control value indicates what component(s) should be active.

Each component's `sensitive to` clause indicates that it should be active when its execution phase begins. As with the instruction interpreter, the machine starts by turning on the `reset` signal. This causes the `fetch` component to activate and sets the instruction pointer to 0. After `reset` turns off, all components are inactive until the `start` signal is asserted. `fetch`'s `sensitive to` clause is the only `sensitive to` clause satisfied by this action, so `fetch` is the only component that activates. All other components have no affect on the concrete state of the architecture. The `fetch` component retrieves the current instruction from memory and places it in the instruction register (`IR`). It also sets the `cntl` signal to `decode`.

```
use work.mm_types.all;
architecture mm_fde of mm is
  component fetch
  port (reset,start : in boolean;
        mem: in memory;
        ip : inout address;
        ir : out word;
        cntl: inout control);
  end component;
  component decode
  port (mem: in memory;
        ip: in address;
        ir: in word;
        ea: out address;
        cntl: inout control);
  end component;
  component execute
  port (mem: inout memory;
        reg: inout registers;
        ea: in address;
        cntl: inout control);
  end component;

  signal CNTL: control;
  signal IP : address;
  signal IR : word;
  signal EA : address;
  signal reg : regfile;

begin
  b1: fetch port map (reset,start,
                      mem,IP.IR,cntl);
  b2: decode port map (mem,IR,EA,CNTL);
  b3: execute port map (mem,reg,EA,CNTL);
end mm_fde;

use work.mm_types.all;
entity fetch is
  port(reset,start : in boolean;
       mem: in memory;
       ip : inout address;
       ir : out word;
       cntl: inout control);
  sensitive to start or reset or
    cntl=fetch;
  modifies ir,cntl;
  requires true;
  ensures
    (reset and ip'post = 0)
    or (not reset and
     ir'post=mem(ip)
     and cntl'post=decode);
end fetch;
```

```
use work.mm_types.all;
entity decode is
  port (mem: in memory;
        ip: in address;
        ir: in word;
        ea: out address;
        cntl: inout control);
  sensitive to cntl=decode;
  modifies ea,cntl;
  requires true;
  ensures
   ((am(ir) = ab and
     ea'post=addr(ir)) or
    (am(ir) = imm and
     ea'post=ip+1) or
    (am(ir) = ind and
     ea'post=mem(addr(ir))) or
    (am(ir) = rel and
     ea'post=ip+addr(ir)))
   and cntl'post=execute;
end decode;

use work.mm_types.all;
entity execute is
  port(mem: inout memory;
       ip: inout address;
       ir: in word;
       reg: inout regfile;
       ea: in address;
       cntl: inout control);
  sensitive to cntl=execute;
  modifies mem,reg,ip,cntl;
  requires true;
  ensures
    (ins(ir) = jump and
     ip'post=addr(ir) and
     cntl'post=fetch)
     or (ins(ir) = load and
       reg(rnum(ir))'post=mem(ea) and
       cntl'post=fetch and
       ((am(ir) = imm and
         ip'post = ip+2) or
        (am(ir) /= imm and
         ip'post = ip+1)))
     or (ins(ir) = store and
       mem(ea)'post=reg(rnum(ir)) and
       cntl'post=fetch and
       ((am(ir) = imm and
         ip'post = ip+2) or
        (am(ir) /= imm and
         ip'post = ip+2)))
     or (ins(ir) = halt and
       cntl'post=halt);
end execute;
```

Figure 16: High level fetch-decode-execute architecture for the Move Machine CPU

The only component whose `sensitive to` clause is satisfied at this point is `decode`. This component calculates the effective address based on the addressing mode specified by the instruction in the `IR` and sets the `cntl` signal to `execute`. The `execute` component then manipulates the registers and memory based on the current instruction. When a `load`, `store` or `jump` instruction is executed, `execute` sets the `cntl` signal to `fetch` which causes the `fetch` component to activate and the process starts again. If the `halt` instruction is processed, `execute` sets `cntl` to `halt`. This makes all three component's `sensitive to` clauses false and the concrete state of the architecture does not change again until something (such as activating `reset`) outside of `mm` changes it.

5 Related Work

5.1 Software Architecture

The research area most closely related to abstract architecture representation in VSPEC is software architecture [8]. Research in this field has led to the development of several architecture description languages, including UniCon [23], WRIGHT [3, 4] and RAPIDE [16, 17]. Each of these languages allow the definition of components and connectors to define a software architecture. This is similar to the VHDL notion of a structural architecture described in this paper.

Shaw's UniCon language [23] is one example of an architecture description language. A UniCon description consists of component and connector definitions. Each of these definitions gives the type (such as Filter or Process for components and Pipe or FileIO for connectors), association units (component players and connector roles) and an implementation for the component or connector. The primary product of the UniCon compiler is Odinfiles, something similar to makefiles that can be used to construct executables for the described architecture. Thus, one of the main products of a UniCon description is a facility that is used to construct an executable version of the described architecture. This is very different from a VSPEC abstract architecture which is used to verify that the class of solutions defined by the architecture implements the requirements specified by the

VSPEC description of the component.

The WRIGHT architecture description language [3, 4] by Allen and Garlan is of particular interest when discussing abstract architectures in VSPEC. A WRIGHT description consists of a collection of components interacting via instances of connector types. Each part of a WRIGHT description is defined using a variant of CSP [11]. Unlike VSPEC's use of CSP to define only communications between components, WRIGHT descriptions use CSP to define the behavior of components as well. WRIGHT's CSP descriptions define the sequence of events that occur in a component or connector. Components and connectors interact when one component/connector *observes* an event *provided* by another. This may cause the second component/connector to *provide* events that cause further interactions. These interactions are all described using CSP.

RAPIDE [16, 17] is an executable architecture description language designed for prototyping architectures of distributed systems. A RAPIDE architecture consists of a set of module specifications (called interfaces), a set of connection rules defining communication between interfaces and a set of formal constraints that define legal patterns of communication. A RAPIDE architecture is executed to produce a partially ordered set of events (poset) that represents the dependencies between events in the architecture. The RAPIDE tools can then verify this poset does not violate the formal constraints defined in the architecture. A major difference between RAPIDE and VSPEC is that VSPEC descriptions are not executable. They are intended for formal analysis.

5.2 Other VHDL-Related Specification Languages

Odyssey Research Associates (ORA) is developing Larch/VHDL, an alternative Larch interface language for VHDL [13]. Larch/VHDL is targeted for formal analysis of a VHDL description and ORA is defining a formal semantics for VHDL using LSL. The LSL representations are used in a traditional theorem prover to verify system correctness. Larch/VHDL annotations are added to a specific VHDL description to represent proof obligations for the verification process. In contrast to this, a VSPEC abstract architecture represents the requirements of a class of solutions that satisfy

a specification (also given in VSPEC).

Augustin and Luckham's VAL [6] is another attempt to annotate VHDL. The purpose of a VAL annotation to a VHDL description is to document the design for verification. VAL provides mechanisms for mapping a behavioral description to a structural description. Two VAL/VHDL descriptions of a design can be transformed into a self-checking VHDL program that is simulated to verify that the two descriptions implement the same function. This differs from VSPEC because it does not allow the description of a class of solutions that implement a specification. Instead, it allows the verification that a structural description correctly maps to a behavioral description for the entity.

5.3 Larch Interface Languages

Larch interface languages have been developed for a variety of programming languages, including LCL [9], Larch/C++ [15] and LM3 [14], interface languages for C, C++ and Modula-3, respectively. Each of these languages allow the description of the pre- and post-conditions for procedures and functions in a sequential programming language. The portions of these languages that allow this type of specification (i.e. `requires`, and `ensures` clauses) are also found in VSPEC where they are used to specify the transformation performed by a single component. However, since C, C++ and Modula-3 are sequential languages, their Larch interface languages do not have to deal with how the Larch-specified procedures and functions interact when two procedures are executing concurrently as is the case with VSPEC entities. At the present time, we are not aware of other work in the Larch community where pre and post-conditions are used to specify the behavior of components in an abstract architecture.

6 Conclusion

6.1 Summary

The ability to evaluate architectural decisions early in the design process enhances overall design quality by allowing architectural errors to be discovered when they are less expensive to fix. Unfortunately, VHDL does not allow evaluation until a simulatable model exists. For many complex systems, simulatable models appear late in the design process making architectural errors difficult to correct. An alternative to simulation for evaluating architectural decisions is formal analysis of abstract architectures at the requirements level. An abstract architecture is a set of interconnected components where the requirements of each component are known but the implementation is not. This paper presented VSPEC's support for describing and evaluating abstract architectures during requirements specification.

A VSPEC abstract architecture is formed by instantiating each component in a VHDL structural architecture with a VSPEC entity. The VSPEC description of an entity includes a pre-condition, post-condition and activation condition that describe the entity's functional requirements. If the current state of the system satisfies the activation condition for one of the components in the abstract architecture, that component's pre-condition must hold and the component must satisfy its post-condition in the next state. A refinement of a VSPEC entity can be compared with the VSPEC specification using weak bisimulation. If some sequence of state changes in the refinement yields the correct single state change in the higher-level description, weak bisimulation holds. This method can be used to formally determine if a VSPEC abstract architecture is a refinement of the VSPEC description of the entity it implements.

6.2 Status and Limitations

VSPEC provides a specification capability most appropriate for high levels of abstraction. It is anticipated that designers will represent system requirements with VSPEC, gradually refining re-

quirements into architectures and eventually a VHDL design. During requirements specification when a designer is defining the essential requirements of a system, VSPEC is useful for evaluating the impact of architectural decisions. When design details are available, VHDL simulation is a more suitable analysis activity. Although VSPEC can model design detail, formal analysis is far less pragmatic than VHDL simulation in such situations.

A potential limitation to the VSPEC approach is verifying the refinement of VSPEC requirements into VHDL design representations. Formalizing the tie between VSPEC and VHDL to support verification and comparison with simulation results is the subject of current investigations. In addition, techniques for automatically synthesizing VHDL from VSPEC are currently under development [21, 20]. Studies of error analysis reports for safety-critical software systems suggest that over 90% of safety related errors arise from incorrect or incomplete specifications, not transformation of requirements into implementations [18]. This suggests that the use of techniques such as those proposed here are warranted even before a complete verification path between VSPEC and VHDL exists.

References

[1] ALEXANDER, P., BARAONA, P., AND PENIX, J. Using Declarative Specifications and Case-Based Planning for System Synthesis. *Concurrent Engineering: Research and Applications 2*, 4 (1994).

[2] ALEXANDER, P., BARAONA, P., AND PENIX, J. Application of Software Synthesis Techniques to Composite Systems. In *Computers in Engineering Symposisum of the ASME ETCE* (Houston, TX, January 1995).

[3] ALLEN, R., AND GARLAN, D. Formalizing Architectural Connection. In *Proc. Sixteenth International Conference on Software Engineering* (May 1994), pp. 71–80.

[4] ALLEN, R., AND GARLAN, D. A Case Study in Architectural Modelling: The AEGIS System. In *Proceedings of the 8th International Workshop on Software Specification and Design* (March 1996).

[5] ASHENDEN, P. *The Designers Guide to VHDL.* Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1996.

[6] AUGUSTIN, L., LUCKHAM, D., GENNART, B., HUH, Y., AND STANCULESCU, A. *Hardware Design and Simulation in VAL/VHDL.* Kluwer Academic Publishers, Boston, MA, 1991.

[7] BARAONA, P., PENIX, J., AND ALEXANDER, P. VSPEC: A Declarative Requirements Specification Language for VHDL. In *High-Level System Modeling: Specification Languages*, J.-M. Berge, O. Levia, and J. Rouillard, Eds., vol. 3 of *Current Issues in Electronic Modeling.* Kluwer Academic Publishers, Boston, MA, 1995, ch. 3, pp. 51–75.

[8] GARLAN, D., AND SHAW, M. An Introduction to Software Architecture. In *Advances in Software Eng. and Knowledge Eng.*, V. Ambriola and G. Tortora, Eds., vol. 2. World Scientific, New York, 1993, pp. 1–39.

[9] GUTTAG, J. V., AND HORNING, J. J. Introduction to LCL, A Larch/C Interface Language. Tech. Rep. 74, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, July 1991.

[10] GUTTAG, J. V., AND HORNING, J. J. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, NY, 1993.

[11] HOARE, C. A. R. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, 1985.

[12] *IEEE Standard VHDL Language Reference Manual.* New York, NY, 1994.

[13] JAMSEK, D., AND BICKFORD, M. Formal Verification of VHDL Models. Technical Report RL-TR-94-3, Rome Laboratory, Griffiss Air Force Base, NY, March 1994.

[14] JONES, K. LM3: A Larch Interface Language for Modula-3. A Definition and Introduction. Version 1.0. Technical Report 72, DEC Systems Research Center, June 1991.

[15] LEAVENS, G. T. Larch/C++ Reference Manual. Available at ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz., 1995.

[16] LUCKHAM, D., KENNEY, J., AUGUSTIN, L., VERA, J., BRYAN, D., AND MANN, W. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering 21*, 4 (April 1995), 315–355.

[17] LUCKHAM, D., AND VERA, J. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering 21*, 9 (September 1995), 717–734.

[18] LUTZ, R. Analyzing software requirements errors in safety-critcal embedded systems. Tech. Rep. 92-27, Department of Computer Science, Iowa State University, 1992.

[19] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[20] PENIX, J., AND ALEXANDER, P. Design representation for automating software component reuse. In *Procedings of the first international workshop on Knowledge-Based systems for the (re)Use of Program libraries* (Nov. 1995).

[21] PENIX, J., BARAONA, P., AND ALEXANDER, P. Classification and retrieval of reusable components using semantic features. In *Procedings of the 10th Knowledge-Based Software Engineering Conference* (Nov. 1995), pp. 131–138.

[22] ROY, J., KUMAR, N., DUTTA, R., AND VEMURI, R. DSS: A Distributed High-Level Synthesis System. *IEEE Design&Test of Computers* (June 1992), 18–32.

[23] SHAW, M., DELINE, R., KLEIN, D., ROSS, T., YOUNG, D., AND ZELESNIK, G. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering 21*, 4 (April 1995), 314–335.

DRAFT

APPENDIX Q:

VSPEC: A declarative specification methodology for system synthesis

Perry Alexander, Philip Baraona, and John Penix
Knowledge-Based Software Engineering Lab
Department of Electrical and Computer Engineering
The University of Cincinnati
Cincinnati, OH USA 45221-0030
Perry.Alexander@UC.edu

Submitted to ICEHDL 95

Abstract

VHDL *provides a means for operationally defining behavior of digital components and for describing composition of components. However, the operational and structural specification techniques require specification of a single design artifact. They do not provide an appropriate means for representing requirements. This paper describes* VSPEC, *a specification language used in conjunction with* VHDL *to axiomatically describing component requirements.* VSPEC *supports specification of constraints and performance requirements as well as the function of a component. Using the* VHDL `architecture` *construct,* VSPEC *can also specify requirements for abstract system architectures.*

1 Introduction

VSPEC is motivated by the need to specify digital system requirements in an implementation independent fashion. Qualitatively, system requirements specify "what" a system should achieve without specifying "how" it should be done. Design specifications are developed from requirements and describe "how" requirements are implemented. Although VHDL [6] supports specification of specific designs, it does little to support requirements specification. In addition, VHDL does not support a consistent representation of constraints.

Lack of requirements and constraint specification has little effect when designing systems requiring few levels of abstraction. However, there is a growing need for systematic design of very large, abstractly defined systems. When starting from extremely high levels of abstraction, the structure of the eventual design is not reflected in requirements. Thus it is difficult to relate an operational specification back to the requirements it is to exhibit. In such situations, explicit requirements and constraint specification allow a designer to work at a high level of abstraction without interference from the details of lower levels.

This paper describes VSPEC, an extension of VHDL which addresses the problem of representing requirements explicitly. In the remainder of this paper, VHDL's method of design specification and VSPEC's additions are presented. The structure of VSPEC and its associated formal basis are presented. How VSPEC and VHDL can be used to specify abstract architectures is presented along with the relationship between VSPEC and algebraic specification.

1.1 VHDL Design Specification

Specification of a design in VHDL involves 3 basic constructs: (1) the `entity` specifies the interface of a system; (2) the `architecture` specifies the behavior and/or structure of a system; and (3) the `configuration` associates a specific `architecture` with an `entity`. The designer specifies a device interface using the `entity` construct, develops one or more structural or behavioral descriptions using the `architecture` and selects a specific implementation for the `entity` using the `configuration` construct.

Each `architecture` associated with an `entity` represents a potential design at some level of abstraction. Structural specifications indicate how components are composed to construct a solution. Behavioral specifications describe the behavior of a solution using an Ada-like programming language. In both cases, specific candidate designs are represented. A specific design is selected by comparing the behavior of that design with the set of system requirements.

Representation of system requirements in VHDL is restricted to an operational style - a "program" is written that describes an artifact having desired characteristics. Although the operational style is an excellent

means for describing specific designs, it is not ideal for describing system requirements for several reasons.

1. It forces representation of a specific design, thus introducing implementational bias.
2. It does not adapt easily to representation of performance constraints.
3. Unimportant characteristics are indistinguishable from required features of the design.
4. Users must deal with unnecessary detail.

Figure 1a is an example VHDL `entity` representing a component that searches a collection of records for a specific record. Note there is no indication of what the component must accomplish or what performance constraints exist for it. The result is a black-box view of the component with no indication of requirements, as shown in Figure 1b. An architecture can be developed, but such an architecture exhibits the negative characteristics discussed.

2 VSPEC Requirements Specification

A solution to requirements representation in VHDL is VSPEC, a Larch interface language [3] developed for VHDL synthesis. The Larch family of specification languages consists of a collection of application specific interface languages and a common shared language. Each interface language defines sets of specification primitives containing useful constructs in a target application language. The shared language serves two purposes. First, it provides a target formal system for translating interface specifications. Second, it provides a language for writing auxiliary specifications and handbooks of common components.

The traditional shared language is a first order algebraic language call LSL. In VSPEC, the primary shared language is REFINE [1], due to its support for transformation and synthesis, its formal basis, and its potential for execution.

Figure 2a shows the VSPEC representation for the same search as the VHDL `entity` in Figure 1. The added clauses specify input conditions, output conditions and constraints. Figure 2b shows a graphical representation of the same information. The VSPEC definition indicates that `Vcc` must be less that or equal to 5 and that the area $(x \times y)$ must be less than 0.3. No constraints are place on heat dissipation (`H`), clock speed (`Clk`) or timing.

The specification associated with Figure 2 avoids many of the problems with the operational specification style. A search routine is specified independently of any implementation by the `ensures` clause. Only characteristics necessary for specifying a search are included. Constraints are clearly specified in the `constrained by` clause and do not interfere with the functional specification. The designer need not be concerned with the details of the search algorithm at the requirements level.

3 The VSPEC `entity`

All VSPEC annotations affect only the VHDL `entity` structure. No changes are made to `architecture` structures or any other VHDL structure. VSPEC clauses are grouped into four broad classes: (1) those that define a devices function; (2) those that define internal state variables; (3) those that define constraints; and (4) those that relate VHDL data structures to formal representations.

3.1 VSPEC Clauses and Logic

VSPEC is a collection of keywords followed by logical sentences. The keywords indicate what requirement each logical sentence specifies. Each logical setence is written in typed first-order predicate calculus. Extensions to the logic allow use of sets and sequences in specifications. The logic follows the basic syntax of REFINE , the language used for system synthesis, to support easy translation and some degree of execution.

There are six basic VSPEC clauses:

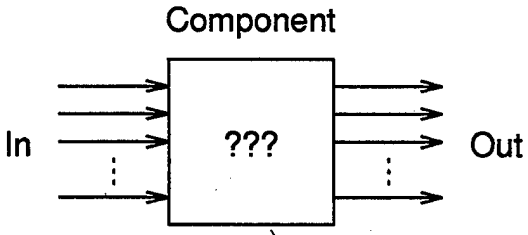
- `requires` – specifies sufficient conditions on inputs and state for `entity` execution
- `ensures` – specifies necessary conditions on outputs and state following `entity` execution
- `constrained by` – specifies non-functional performance constraints
- `modifies` – specifies what the `entity` may alter
- `based on` – associates VHDL data types with REFINE definitions
- `state` – defines a collections of variables that represent the `entity`'s internal state

VSPEC clauses may only access variables and signals defined in an `entity port`, the `state` clause or quantified in a logical expression. VSPEC is strongly typed and all variables must have an associated type, including those bound by quantifiers. Although REFINE allows type inferencing, VSPEC does not.

All VSPEC clauses are optional. Only the `based on` clause may appear more than once in an `entity`. The format of the `requires`, `ensures`, and `constrained by` clauses is a keyword followed by a logical expression and a semicolon.

<keyword> *<logical-expression>* ";"

```
entity search is
  port (input: in array of element;
        k: in keytype;
        output: out element);
end search;
```

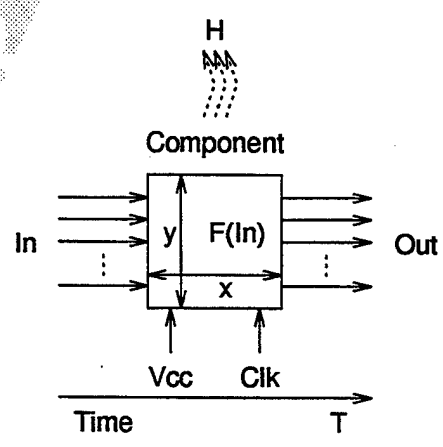


a) b)

Figure 1: A VHDL entity describing a record search.

```
entity search is
  port (input: in array of element;
        k: in keytype;
        output: out element);
  modifies output;
  requires true;
  ensures
    output = e <=> key(e)=k and
                   e in input
  constrained by
    power =< 5 and
    area =< .3
end search;
```

a)



b)

Figure 2: A VSPEC entity describing a record search.

The format of the **state** and *modifies* clauses is a keyword followed by a collection of variables, optionally typed.

<*keyword*> <*variable*>[, <*variable*>] ";"

The format of the **based on** clause is a type name followed by the **based on** keyword and a logical expression.

<*type*> **based on** <*logical-expression*> ";"

3.2 Functional Requirements

The functional requirements of a VSPEC **entity** are defined using the **requires** and **ensures** clauses. The **requires** clause specifies a logical expression, $I(x)$, that must be true for the **entity** to perform its operation. The **ensures** clause specifies necessary state conditions, $O(x, z)$, resulting from **entity** execution given a particular input. Formally, any function implementing an **entity** must obey the condition:

$$\forall x : D \bullet I(x) \Rightarrow O(x, F(x)) \qquad (1)$$

3.2.1 The requires Clause

The **requires** clause, $I(x)$, is a logical expression defined over all ports, signals and variables that may provide input to the transform. $I(x)$ is true when x is a valid input. $I(x)$ is a precondition for **entity** execution. When it is true, the **entity** must produce valid output.

3.2.2 The ensures Clause

The **ensures** clause, $O(x, z)$, is a logical expression defined over all ports, signals and variables. $O(x, z)$ is true when z is a valid output given x as input. $O(x, z)$ is a postcondition for **entity** execution and states necessary conditions placed on **entity** outputs and state variables.

3.3 Constraints

Constraints express characteristics an **entity** must exhibit that are not a part of its function. For example, heat dissipation constraints frequently affect selection of valid designs, but heat is a side effect of the technology. It has little to do with input and output relationships.

Although constraints do not affect function, they are critical in hardware system design. In VSPEC there are two sources of constraint. The first is the **constrained by** clause that specifies several performance constraints common in hardware design. The second is the **modifies** clause that limits what the **entity** can alter in performing its function.

3.3.1 The constrained by Clause

The **constrained by** clause is a conjunction of predefined variables and relations with fixed values. VSPEC currently supports providing constraint information for heat dissipation, area, clock speed, power consumption and pin-to-pin timing. To specify constraint, one chooses a constraint type and uses it in a relation. For example, to specify heat dissipation less than 1 watt and power consumption less than 10 watts, the logical sentence **heat =< 1 and power =< 10** is included in the **constrained by** clause.

Timing requires a somewhat more complicated representation. Here one specifies an interval between two pins, then relates that interval to a constant time. For example, **(a<->b) =< 10** specifies that the time between a signal arriving at port **a** and port **b** producing a signal must be lass than 10.

3.3.2 The modifies Clause

The **modifies** clause specifies a collection of ports, signals and variables that may be modified by the entity. The **modifies** clause indicates what effects and side effects are allowed. Only outputs may be specified in a **modifies** clause. Of particular interest is the ability to specify the direction of **buffer** type ports.

3.4 Abstract Data Types

The semantics of VHDL data types must be defined before reasoning about their properties is possible. Elemental data types such as **integer** and **bit** have definitions loaded as a part of the VSPEC system. Thus, when using a basic VHDL type, the semantics of that type are present by default.

3.4.1 The based on Clause

User defined data types such as arrays and records must be defined as a part of the definition process because they cannot be defined *a priori*. This is accomplished using the **based on** predicate. The logical expression defined in a **based on** clause defines the semantics of a user defined type. To support this specification process, VSPEC include standard schemas for defining sets, sequences, arrays and tuples. These schemas are used in conjunction with parameter morphism to define associated VHDL types specific to user needs.

3.5 System State

The notion of system state is typically not supported directly by axiomatic specification techniques. A computation unit is defined by a transform that relates inputs to outputs. Thus, to include state in a

specification it must be specified as an input to the transform. However, specification of state-based systems is natural to hardware designers and suggesting that state representation be an input to the VHDL **entity** is not natural. Using the two-tiered specification approach state can be managed by: (a) supporting the definition of local state variables; and (b) using state maintaining features of **port** signals. Instead of specifying a function that maps input signals defined in the **port** definition to outputs in the same **port** definition, specify a function that maps inputs and state maintaining objects to outputs and state maintaining objects.

3.5.1 The state clause

The **state** clause is a collection of variables that store state within a VSPEC **entity**. Like VHDL variables and signals, these variables maintain their values from one invocation of the **entity**. All **state** variables are defined locally and are not visible outside the entity.

3.5.2 Ports

Variables defined an **entity**'s **port** definition may maintain their state. Variables of type **buffer** may be inputs or outputs and are not re-initialized unless a signal of some type is driving them. Variables of type **out** and **inout** also maintain their state.

4 Generic Architectures in VSPEC

VSPEC supports representation of high level, abstract architectures using the **architecture** construct from VHDL. No modifications or annotations are necessary - simply specify **entity** structures accessed by the **architecture** using VSPEC.

Figure 3 represents a two component architecture for solving the element search problem. The **architecture batch-seq** represents a two step solution of sorting the input list and using a binary search to find the desired record. Although the requirements of the sorting algorithm are specified, no algorithm is presented. Thus, the designer may instantiate the sort with any appropriate algorithm. Application of such an architecture represents an iterative refinement process common to design activities. Additionally, VSPEC is adept at representing such refinements where an operational language may fall short.

5 VSPEC and Algebraic Specification

Any VSPEC definition can be transformed into a formal definition. The form of this definition is an algebraic specification based on an extension of domain theories as defined in CYPRESS [7] and KIDS [9, 8]. The basic form of a domain theory is a tuple consisting of the function domain (D), range (R), input precondition (I(x:D)) and output postcondition (O(x:D,z:R)) commonly referred to as a *DRIO* model. The *DRIO* model for any VSPEC **entity** can be constructed using the following rules:

$D = t_1 \times t_2 \times \ldots \times t_n$ where t_k is the sort representing the type associated with an **in**, **inout**, or **buffer** ports, or a state variable

$R = t_1 \times t_2 \times \ldots \times t_m$ where t_j is the sort representing the type associated with an **out**, **inout**, or **buffer** port listed in the **modifies** clause, or a state variable listed in the modifies clause

$I(x : D) = I_v(x : D)$ where $I_v(x : D)$ is the logical sentence defined by the **requires** clause

$O(x : D, z : R) = O_v(x : D, z : R)$ where $O_v(x : D, z : R)$ is the logical sentence defined by the **ensures** clause

Additionally, constraints must be defined as a part of the algebraic statement. The simplest means of accomplishing this is to simply include predicates representing constraints in the output function of the DRIO. However, constraints are not functional. Specifying constraints in their own clause is an attempt to separate constraint from function. Additionally, constraints in their current form do not depend on variables defined in the **entity**[1]. Thus, constraints are added to the DRIO model through a specification morphism that adds logical representations of constraints. The DRIO model becomes a DRIOC model.

$C(c_1 : C_1, \ldots, c_n : C_n) = C_v(c_1 : C_1, \ldots, c_n : C_n)$ where c_k is a constraint variable such as heat or area, C_k is a sort associated with a constraint variable and C_v is the logical expression defined in the **constrained by** clause

The goal of the design activity is to find and architecture that performs the transform $F : D \rightarrow R$ such that:

$$\forall x : D \bullet I(x) \Rightarrow O(x, F(x)) \qquad (2)$$

Thus, the goal of the synthesis activity is generation of a transform mapping the current state and inputs into the next state and outputs such that the output condition is satisfied.

[1] A more complex constraint model could certainly include variables and signals. Our current constraint model does not allow this.

```
architecture bat-seq of search is
  component sorter
    port (input: in array of element;
          output: out array of element);
  component bin_search
    port (input: in array of element;
          key: in integer;
          value: out element);
begin
  b1: c1 port map(x,y);
  b2: c2 port map(y,z);
end bat-seq;
```

```
entity sort is
  port (input: in array of element;
        output: out array of element);
  modifies output;
  ensures bag(input) = bag(output) and
          sorted(output)
end sort;

entity bin_search is
  port (input: buffer array of element;
        k: in integer;
        value: out element);
  modifies out;
  requires sorted(input);
  ensures
    (fa e:element)
      output = e <=> key(e)=k and
                     e in input
end bin_search;
```

Figure 3: VSPEC representation of a search architecture using a batch sequential approach. The original list is sorted and a binary search finds the desired object from the resulting list.

6 Related Work

As VSPEC is a Larch interface language for VHDL it borrows from the construction of other interface languages. Specifically, VSPEC is styled after the LM3 Larch interface language for Modula-3 [5]. Odyssey Research Associates is currently developing an alternative Larch interface language for VHDL [4]. This language does not support representation of constraints and is targeted for formal analysis rather than synthesis. ORA's interface language also differs in its implementation of time. An absolute time based temporal logic is used in specifying the function of an `entity`. Thus one can specify that a predicate becomes true at a specific time using the notation "$P(x)@t$".

Another attempt to annotate VHDL is VAL [2]. VAL annotates all aspects of the VHDL design. All signals in the namespace of the VHDL representation are in the namespace of the VAL annotation. Thus, VAL annotates specific VHDL designs rather than represent requirements. ORA's interface language is similar in this respect, but does support separate requirements definitions.

7 Future Work

Current VSPEC research involves pursuing domain specific support for specification activities and support for formal synthesis. An important aspect of any Larch language is its associated handbook. A handbook is simply a collection of reusable theories defined in the shared language. Handbook theories represent commonly used structures, algorithms and characteristics as well as domain specific information. For VHDL we are implementing theories to represent standard VHDL types, low level logic functions and conversion routines. In addition, we are working on libraries to support specifications involving signal attributes such as `event`, `stable`, and `delay`. Theories for pin-to-pin timing, heat dissipation, power consumption, area and clock speed have been implemented to support constraint checking during the design process.

The isomorphic relationship between VSPEC and algebraic specifications is being used to exploit work in formal synthesis, specifically, developing morphisms between algorithms [10]. This involves development and implementation of theories useful in constructing multicomponent systems such as the batch sequential search algorithm appearing earlier in this paper.

8 Acknowledgments

Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615-93-C-1316. The authors wish to thank Wright Labs and ARPA for their continuing support.

References

[1] L. Abraido-Fandino. An overview of refine 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering*, Madrid, Spain, April 1987.

[2] L. Augustin, D. Luckham, B. Gennart, Y. Huh, and A. Stanculescu. *Hardware Design and Simulation in VAL/VHDL.* Kluwer Academic Publishers, Boston, MA, 1991.

[3] J. Guttag and J. Horning. *Larch: Languages and tools for formal specification.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1993.

[4] D. Jamsek and M. Bickford. Formal Verification of VHDL Models. Technical Report RL-TR-94-3, Rome Laboratory, Griffiss Air Force Base, NY, March 1994.

[5] K. Jones. LM3: A Larch Interface Language for Modula-3. Technical Report 72, DEC Systems Research Center, Palo Alto, CA, 1991.

[6] D. Perry. *VHDL.* McGraw-Hill, New York, NY, 1st edition, 1991.

[7] D. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, Sept. 1985.

[8] D. Smith. Algorithm Theories and Design Tactics. *Science of Computer Programming*, 14:305–321, 1990.

[9] D. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, Sept. 1990.

[10] D. Smith. Classification approach to design. Technical Report KES.U.93.4, Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA, November 1993.

APPENDIX R:

VSPEC: A declarative specification methodology for system requirements

Phillip Baraona, John Penix, and Perry Alexander
Knowledge-Based Software Engineering Lab
Department of Electrical and Computer Engineering
The University of Cincinnati
Cincinnati, OH USA 45221-0030
Perry.Alexander@UC.edu

Abstract

Systems engineering of computer-based systems demands explicit representation of functional requirements as well as constraints at each level of design abstraction. However, traditional design representation languages such as VHDL *and* VERILOG *do not support requirements representation independent from implementation. This work presents a axiomatic specification language designed to support requirements representation.* VSPEC *annotates* VHDL *entity structures supporting declarative specification of input preconditions, output postconditions and performance constraints as a part of the design representation. The declarative nature of the specification supports requirements definition independent of design representation.*

1 Introduction

It is commonly understood that engineering is a requirements driven activity. Problem requirements are stated and the engineering goal is to produce an artifact satisfying those requirements. Requirements can be broadly categorized into two classes: (1) functional requirements; and (2) constraints. Although the distinction between these two classes is frequently debated, functional requirements describe the intended transformation from input to output while constraints describe other non-functional restrictions placed on the solution. Both functional requirements and constraints must be represented and accounted for in a successful systems engineering activity.

VHDL [1] is a widely accepted design specification language for digital systems. It supports representation of artifacts at multiple levels of abstraction as well as providing both behavioral and structural descriptions. Unfortunately, VHDL supports only an operational specification style and provides no standard means for representing constraints. Thus, when used at the requirements level, VHDL forces the user to make implementation decisions early in the design process. As the desired result of requirements analysis is a description of "what" without regard to "how", VHDL is not an appropriate requirements representation language. In addition, constraint information frequently used to choose between design alternatives is not explicitly represented.

VSPEC is a Larch[2] interface language for VHDL that supports declarative specification of both functional requirements and constraints. VSPEC defines functional requirements using an input precondition and output postcondition defined over the ports and internal state of a VHDL `entity`. VSPEC defines constraint information using standard representations of heat dissipation, clock speed, delay time, area and power consumption limits. In addition, other constraint types may be defined by the user.

This paper describes the VSPEC language and how it is used to define systems level requirements. A brief presentation of VHDL is given and problems identified. The basic structure of VSPEC is then described followed by specifics of language constructs. Also presented is a means for using VSPEC and structural VHDL to define high-level architectures, thus supporting high level decomposition. Finally, the role of VSPEC in the design process is shown along with examples of its use.

2 VHDL Design Specification

Specification of a design in VHDL involves 3 basic constructs: (1) the **entity** specifies the interface of a system; (2) the **architecture** specifies the behavior and/or structure of a system; and (3) the **configuration** associates a specific **architecture** with an **entity**. The designer specifies a device interface using the **entity** construct, develops one or more behavioral or structural descriptions using the **architecture** and selects a specific implementation for the **entity** using the **configuration** construct.

Each **architecture** associated with an **entity** represents a potential design at some level of abstraction. Behavioral specifications describe the behavior of a solution using an Ada-like programming language. Structural specifications indicate how components are composed to construct a solution. In both cases, specific candidate designs are represented. A specific design is selected by comparing the behavior of that design with the set of system requirements.

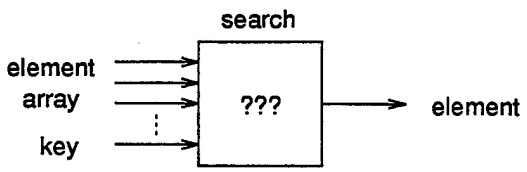
Representation of system requirements in VHDL is restricted to an operational style - a "program" is written that describes an artifact having desired characteristics. Although the operational style is an excellent means for describing specific designs, it is not ideal for describing system requirements for several reasons.

1. It forces representation of a specific design, thus introducing implementational bias.
2. It does not adapt easily to representation of performance constraints.
3. Implementation/representation specific details are indistinguishable from required features of the design.
4. Users must deal with unnecessary detail.

Figure 1a is an example VHDL **entity** representing a component that searches a collection of records for a specific record. Note there is no indication of what the component must accomplish or what performance constraints exist for it. The result is a black-box view of the component with no indication of requirements, as shown in Figure 1b. An architecture can be developed, but such an architecture exhibits the negative characteristics discussed.

```
entity search is
  port (input: in array of element;
        k: in keytype;
        output: out element);
end search;
```

a)



b)

Figure 1: A VHDL **entity** describing a record search. Note that the **entity** defines only the interface. The **architecture** describes the function operationally.

3 VSPEC Requirements Specification

A solution to requirements representation in VHDL is VSPEC, a two-tiered specification language developed for VHDL synthesis. VSPEC is designed using concepts developed for Larch [2] interface languages for software specification. The Larch family of specification languages consists of a collection of application specific interface languages and a common shared language. Each interface language defines sets of specification primitives containing useful constructs in a target application language. The shared language serves two purposes. First, it provides a target formal system for translating interface specifications. Second, it provides a language for writing auxiliary specifications and handbooks of common components.

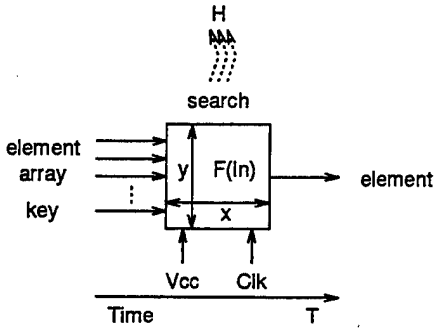
The traditional shared language is a first order algebraic language call the Larch Shared Language (LSL) [3]. In VSPEC, the primary shared language is REFINE[4, 5], due to its support for transformation and synthesis, its formal basis, and its potential for execution.

Figure 2a shows the VSPEC representation for the same search as the VHDL **entity** in Figure 1. The added clauses specify input conditions, output conditions and constraints. Figure 2b shows a graphical representation of the same information. The VSPEC

definition indicates that power consumption must be less that or equal to 5 mW and that the size $(x \times y)$ must be less than $5 \times 3\mu m^2$. No constraints are place on heat dissipation (H), clock speed (Clk) or timing.

```
entity search is
  port (input: in array of element;
        k: in keytype;
        output: out element);
  modifies output;
  requires true;
  ensures
    output = e <=> key(e)=k and
                   e in input
  constrained by
    power <= 5 mW and
    size <= 3 um * 5 um
end search;
```

a)



b)

Figure 2: A VSPEC **entity** describing a record search. The functional requirements and constraints are explicitly represented as a part of the **entity** construct.

The specification associated with Figure 2 avoids many of the problems with the operational specification style. A search routine is specified independently of any implementation by the **ensures** clause. Only characteristics necessary for specifying a search are included. Constraints are clearly specified in the **constrained by** clause and do not interfere with the functional specification. The designer need not be concerned with the details of the search algorithm at the requirements level.

4 The VSPEC entity

All VSPEC annotations affect only the VHDL **entity** structure. No changes are made to **architecture** structures or any other VHDL structure. VSPEC clauses are grouped into four broad classes: (1) those that define a devices function; (2) those that define internal state variables; (3) those that define constraints; and (4) those that relate VHDL data structures to formal representations.

4.1 VSPEC Clauses and Logic

VSPEC is a collection of keywords followed by logical sentences. The keywords indicate what requirement each logical sentence specifies. Each logical sentence is written in typed first-order predicate calculus. Extensions to the logic allow use of sets and sequences in specifications. The only variables allowed in each clause are: (1) **ports**; (2) variables defined in the **entity**s **state** clause; and (3) variables defined by quantifiers in the sentence. Both **port** and **state** variables are assumed to be universally quantified. The only exception to this rule is the **constrained by** clause where variables defined in constraint theories are used exclusively.

There are six basic VSPEC clauses:

- **requires** – specifies sufficient conditions on inputs and state for **entity** execution
- **ensures** – specifies necessary conditions on outputs and state following **entity** execution
- **constrained by** – specifies non-functional performance constraints
- **modifies** – specifies what the **entity** may alter
- **based on** – associates VHDL data types with REFINE definitions
- **state** – defines a collections of variables that represent the **entity**'s internal state
- **includes** – specifies that a shared language file containing data types and functions is used in the definition
- **assumes** – specifies assumptions made in defining the device[1]

VSPEC clauses may only access variables and signals defined in an **entity port**, the **state** clause or quantified in a logical expression. VSPEC is strongly typed and all variables must have an associated

[1]This clause is not implemented in the current language parser, but will be included in a later release

type, including those bound by quantifiers. Although REFINE allows type inferencing, VSPEC does not.

Logical statements in VSPEC are designed to mimic as much as possible the syntax of VHDL. This supports ease of use by VHDL users and achieves the language specific goals of a Larch interface language. For example, numerical constants follow VHDL format, logical connectives use their English names, and predicates defined on signals follow the `<signal>'<property>` convention defined for VHDL. This changes the standard Larch `<variable>'` representation for the post execution value of `<variable>` to `<variable>'post`.

4.2 State-Based Specification

The VSPEC model uses a classic state-based specification approach. The notion of system state is typically not supported directly by axiomatic specification techniques. A computation unit is defined by a transform that relates inputs to outputs. Thus, to include state in a specification it must be specified as an input to the transform. However, specification of state-based systems is natural to hardware designers and suggesting that state representation be an input to the VHDL **entity** is not natural. Using the two-tiered specification approach state can be managed by: (a) supporting the definition of local state variables; and (b) using state maintaining features of **port** signals. Instead of specifying a function that maps input signals defined in the **port** definition to outputs in the same **port** definition, specify a function that maps inputs and state maintaining objects to outputs and state maintaining objects.

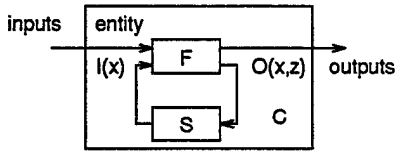


Figure 3: State-based specification model that forms the basis of VSPEC requirements definition.

The goal is specifying a function that accepts input values and the current state and generates output and a new state. To achieve this, VSPEC specifies an input precondition over inputs and state, and an output postcondition over outputs and state. Figure 3 shows these relationships graphically. F is the function of the component, S stores the internal state, and C defines constraints. $I(\bar{x})$ defines a precondition on inputs and state while $O(\bar{x}, \bar{z})$ defines a postcondition on outputs and state given an input. Finally, $C(\bar{c})$ defines a set of constraints the device must operate under.

A device's interface is defined by the VHDL **entity** construct. VSPEC uses these definitions in its clauses to reference these signals rather than redefining the interface. VSPEC defines a devices function by providing S and stating $I(\bar{x})$ and $O(\bar{x}, \bar{z})$. Finally, VSPEC defines constraints by defining predicates over $\bar{c}$, a constraint variable set.

4.3 Internal State

The **state** clause defines a collection of variables and initial values defining the internal state of a component. These variables are not visible outside the **entity**. State variables maintain their values between **entity** invocations. As with any VSPEC symbol, the undecorated state variable name indicates the value before invocation and the name decorated with **'post** indicates the value after invocation. Thus, values before and after invocation are accessible in the same definition.

It is important to note that VHDL ports also maintain their values between **entity** invocations. However, **ports** are visible outside the **entity** and need not be defined in the **state clause**. The **state** clause defines only new variables necessary for internal state components. It is possible (even common) for components having no **state** clause to be state based using only port values as state. The entire state of a component is the complete set of state variables and ports. Like state variables, the **'post** attribute supports accessing both a port's pre-invocation and post-invocation values.

4.4 Functional Requirements

The functional requirements of a VSPEC **entity** are defined using the **requires** and **ensures** clauses. The domain, D, of F is the set of all finite vectors consisting of: (1) ports providing input; and (2) state variables. The range, R, of F is the set of all finite vectors consisting of: (1) ports generating output; and (2) state variables. The direction indicators used in VHDL **port** definitions and the **modifies** clause determine what ports and state variables are appropriate for D and R. Note that a port or state variable may appear in both D and R.

The **requires** clause specifies a logical expression, $I(\bar{x})$, that must be true for the **entity** to perform its operation. The vector $\bar{x}$ is an element of D. The **ensures** clause specifies necessary post-conditions, $O(\bar{x}, \bar{z})$, resulting from **entity** execution given a particular input. The vector $\bar{z}$ is an element of R. Any function, F, implementing an **entity** must obey the condition specified in Equation 1. The pre- and post-conditions, I and O, defined by the **ensures** and **requires** clauses represent the **entity**'s functional requirements.

$$\forall x : D \bullet I(\bar{x}) \Rightarrow O(\bar{x}, F(\bar{x})) \qquad (1)$$

Equation 1 defines a synthesis goal considering only functional requirements.

4.5 Constraints

Constraints express characteristics an **entity** must exhibit that are not a part of its function. For example, heat dissipation constraints frequently affect selection of valid designs, but heat is a side effect of the technology. It has little to do with input and output relationships.

Although constraints do not affect function, they are critical in system design. In VSPEC, two clauses are used to represent constraints. The first is the **constrained by** clause that specifies several performance constraints common in hardware design. The second is the **modifies** clause that limits what the **entity** can alter in performing its function. The **constrained by** clause is a conjunction of predicates defined over a constraint variable set, $\bar{c}$. Adding constraints to Equation 1 results in the new synthesis goal for F shown in Equation 2. Note that $C(\bar{c})$ is the conjunction of predicates specified in the **constrained by** clause.

$$\forall x : D \bullet I(\bar{x}) \Rightarrow O(\bar{x}, F(\bar{x})) \wedge C(\bar{c}) \qquad (2)$$

Equation 2 defines a more realistic synthesis goal adding constraints to the functional requirements. The variables in $\bar{c}$ are defined by underlying constraint theories and are not defined as a part of each **entity**. When specifying an **entity**, constraint variables are inherited from the underlying constraint theory. The current default constraint set supports representation of power consumption, heat dissipation, clock speed, pin-to-pin timing and area. Users may define additional constraints as needed using REFINE to define theories. The new theory is added using the **includes** clause to load the definition.

4.6 Data Types

The semantics of VHDL data types must be defined before reasoning about their properties is possible. Elemental data types such as **integer** and **bit** have definitions loaded as a part of the VSPEC system. Thus, when using a basic VHDL type, the semantics of that type are present by default. VSPEC generates formal definitions of **RECORD** and **ARRAY** types using standard **tuple** and **sequence** constructs from REFINE.

5 Architectures in VSPEC

VSPEC supports representation of high level, abstract architectures using the **architecture** construct from VHDL. A high-level architecture is a collection of interconnected component definitions. Each component is instantiated appropriately for a given problem. High-level architectures provide skeletal solutions for commonly used system architectures – their use is fundamental in complex system design. Taking a single VSPEC **entity** and using a VHDL **configuration** statement to assign a high-level architecture to it supports incremental design activities.

Structural VHDL defines systems by indicating interconnection between components. Within a structural VHDL **architecture**, components are identified and generic parameters instantiated. These components are then used to produce a netlist specifying component interconnection. This interconnection specification is declarative because it simply specifies what components are used and how then are connected. Rather than extend the structural VHDL **architecture** to represent high-level architectures, VSPEC uses it to define interconnections between specified components. VSPEC provides requirements definitions for any or all components in the architecture.

Figure 4 represents a two component architecture for solving the element search problem specified earlier. The **architecture batch-seq** represents a two step solution of sorting the input list and using a binary search to find the desired record.

The **architecture** references two components, a **sorter** and a **bin_search**. In typical structural VHDL, structural or behavioral descriptions exist for each component either decomposing the solution further or describing a behavioral solution. If the **entity** representation for each component is annotated with VSPEC, a third option is possible. No **architecture** is associated with either **entity**,

thus specifying only component requirements. Now three specification options exist: (1) requirements for each component may be specified using VSPEC; (2) the implementation of each component may be specified using structural VHDL; or (3) the behavior of each component may be specified using behavioral VHDL. Realistically, all three will be used at any given time due to varying stages of component design.

Although each component's requirements are specified, no component algorithms or assemblies are presented. However, this new requirements specification exists at a lower level of abstraction, because some structural detail has been added, excluding some potential solutions and decreasing the overall abstraction level. Application of such an architecture represents an incremental refinement process common to design activities. By assigning **bat-seq** to the **entity** from Figure 2, using a **configuration** statement, a requirements decomposition is performed. The resulting **architecture** specifies requirements and interconnections for components and an obligation exists to verify the resulting decomposition is correct with respect to the **entity**'s original requirements.

In addition to functional requirements, constraints play a large role in the **architecture** specification. Constraints are also "decomposed" across collections of components. The simplest example of this activity is budgeting power consumption, weight or heat dissipation. When budgeting, a fraction of the value being constrained is assigned to each component in such a way that the initial constraint is met. With heat dissipation and power, the sum of component constraint limits must not exceed the initial constraint limit.

Although budgeting is common and useful, not all constraints can be managed in this straightforward fashion. Maintainability, reliability, and reuseability are examples of constraints that cannot be budgeted across component collections. However, the methodology continues to apply when a constraint model is developed and used to determine when the decomposition meets the initial constraint limit. Although developing a safety metric, for example, may be a difficult task, if one is developed, it can be incorporated easily into the VSPEC model.

Module fan-out is an example maintainability constraint that cannot be budgeted. Fan-out is the number of modules a single module decomposes

```
architecture bat-seq of search is
  component sorter
    port (input: in array of element;
          output: out array of element);
  component bin_search
    port (input: in array of element;
          key: in integer;
          value: out element);
begin
  b1: c1 port map(x,y);
  b2: c2 port map(y,z);
end bat-seq;

entity sort is
  port (input: in array of element;
        output: out array of element);
  modifies output;
  ensures bag(input) = bag(output) and
          sorted(output)
  constrained by
    power <= 3 mW and
    size <= 1 um * 2 um
end sort;

entity bin_search is
  port (input: buffer array of element;
        k: in integer;
        value: out element);
  modifies out;
  requires sorted(input);
  ensures
    (forall e:element)
      output = e <=> key(e)=k and
                     e in input
  constrained by
    power <= 1 mW and
    size <= 1 um * 2 um
end bin_search;
```

Figure 4: VSPEC representation of a search architecture using a batch sequential approach. The original list is sorted and a binary search finds the desired object from the resulting list.

into. If fan-out is high, then the complexity of the decomposition may be too high to manage effectively. A VSPEC model of fan-out adds a `fanout` predicate to the `constrained by` clause. Specifying `fanout(f) < 10` says the fan-out of the component must be less than 10. The underlying fan-out theory expresses that fan-out is the number of submodules a component has. This provides a means for checking fan-out in an evolving system.

6 Design Process

Using VSPEC and the VHDL `architecture` incremental design results in a tree generated by specialization activities. Consider the earlier search problem. In this design activity, the initial requirements are shown in Figure 2. These requirements completely define the design problem specifying both function and constraint.

When the high-level architecture, `bat-seq` (Figure 4) is associated with the initial requirements, an incremental design decision is represented. This decision represents initial problem decomposition into interconnected search and sort components. These components each have their associated requirements and constraints. At this point in the design process, explicit constraint representation allows the user to check constraints. Namely, that power does not exceed 5 mW and size does not exceed 15 um. Naive constraint theories indicate that constraint budgets do not exceed high level constraints. Without explicit representation, such verification would not be possible. Although these theories are naive, more realistic theories are easily encoded as REFINE specifications.

Functional requirements are also checked using pre- and post-condition comparison. In this case, I and O from the architecture match their corresponding specifications in the system description. Unfortunately, this will rarely be the case, thus requiring more complex checks. However, the requirements are represented explicitly in the design representation and are available for verification.

Assume finally that each component is expressed using behavioral VHDL and fabricated resulting in two hardware components. Fabrication results may be verified independently with some confidence their composition will meet requirements. Additionally, if constraints cannot be met, trade-off decisions may be explored and verified within the context of the entire problem.

7 Related Work

7.1 Larch

VSPEC is based on Larch's two-tiered specification approach and is a Larch Interface Language. VSPEC differs from existing Larch languages in its use of REFINE as its shared language. The Larch Shared Language [3] is a first order, algebraic language while REFINE is a broad-spectrum language that is both executable and formal. REFINE is used because its environment supports software synthesis while Larch is primarily useful for verification. VSPEC's syntax is derived primarily from the Modula-3 interface language, LM3 [6].

7.2 VHDL Annotation Language (VAL)

VSPEC is frequently compared to the VHDL Annotation Language (VAL) [7]. VAL is an annotation language used to describe pre- and post-conditions on VHDL input and output streams. In this respect, VAL and VSPEC are quite similar. However, several critical differences exist. First, VAL annotations translate into VHDL `assert` statements. An `assert` statement is a boolean valued function that causes an event to occur when triggered. The `assert` is much like an exception in a traditional programming language and is used for similar purposes. Once transformed into `assert` statements, the VAL model is simulated on input streams and the result compared to simulation of VHDL code for the same module. VSPEC has support for execution, but this is not its primary purpose. The logic used is not restricted to an executable subset. More importantly, the logic can be manipulated formally.

VAL supports annotation of behavioral and structural VHDL as well as the `entity` structure. Thus, VAL is an annotation language or design description language rather than strictly a requirements language.

Finally, VAL does not support constraint representation or checking. In the systems engineering environment, constraints are frequently more difficult to meet than functional requirements. Furthermore, they must be recorded as a part of any requirements specification.

7.3 ORA's Larch/VHDL

ORA is currently developing a Larch/VHDL interface language. [8] In many respects, this language is similar to VAL in its attempt to model entire systems rather than simply modeling requirements. This language is manipulated formally, thus it is being used to define a semantic model for VHDLLike

VAL, ORA's interface language supports only timing constraints and it's usefulness is therefore limited in the systems engineering area. VSPEC differs substantially, supporting only requirements specification and including both function and constraint. VSPEC also models timing as a constraint where ORA's language uses a temporal logic to model timing attributes.

8 Current Status and Future Directions

Currently, an initial Language Reference Manual for VSPEC is being developed. From the VSPEC LRM, a VSPEC parser and partial type checker have been developed using the DIALECT component of the SOFTWARE REFINERY[4]. This parser is available via the world wide web and ftp.

This version of VSPEC is limited to representing digital information as is VHDL. Plans exist to combine VSPEC with the ANAVHDL work underway at the University of Cincinnati. ANAVHDL supports specification of both analog and digital components in the same system. As VSPEC is declarative and most circuit specifications are specified using equations, this combination is quite natural. Open and interesting problems include interfaces between analog and digital components and reconciliation of timing information from the digital and analog worlds.

9 Summary

VSPEC is a Larch interface language for VHDL designed to represent design requirements for synthesis activities. VSPEC design goals center on: (1) requirements representation independent of implementation; and (2) constraint representation. VSPEC adds declarative components that describe a component's functional requirements and constraints. Axiomatic specifications describe functional requirements by defining input pre-conditions and output post-conditions. Predicates defined over constraint variables describe component constraints. VSPEC supports descriptions of high-level architectures using structural VHDL and allows incremental design step representation.

10 Acknowledgments

Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615-93-C-1316. The authors wish to thank Wright Labs and ARPA for their continuing support.

References

[1] D. Perry, *VHDL*, McGraw-Hill, New York, NY, 1st edition, 1991.

[2] John V. Guttag and James J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, New York, NY, 1993.

[3] V. Guttag, J. Horning, and A. Modet, "Report on the Larch Shared Language: Version 2.3", Technical Report 58, DEC Systems Research Center, April 1990.

[4] L. Abraido-Fandino, "An overview of refine 2.0", *in Proceedings of the Second International Symposium on Knowledge Engineering*, Madrid, Spain, April 1987.

[5] Reasoning Systems Inc., Palo Alto, CA, *Refine User's Guide, Version 3.0*, May 1990.

[6] K. Jones, "LM3: A Larch Interface Language for Modula-3. A Definition and Introduction. Version 1.0", Technical Report 72, DEC Systems Research Center, June 1991.

[7] L. Augustin, D. Luckham, B. Gennart, Y. Huh, and A. Stanculescu, *Hardware Design and Simulation in VAL/VHDL*, Kluwer Academic Publishers, Boston, MA, 1991.

[8] D. Jamsek and M. Bickford, "Formal Verification of VHDL Models", Technical Report RL-TR-94-3, Rome Laboratory, Griffiss Air Force Base, NY, March 1994.

APPENDIX S: Application of Software Synthesis Techniques to Composite Systems

Perry Alexander, Philip Baraona, and John Penix
Knowledge-Based Software Engineering Lab
Department of Electrical and Computer Engineering
The University of Cincinnati
Cincinnati, OH USA 45221-0030
Perry.Alexander@UC.edu

Submitted to: ETCE-95 Engineering Software Session

July 15, 1994

Abstract

Prototyping composite hardware/software systems requires synthesis of hardware, software and communications protocols. Capabilities exist to synthesize ASIC designs from a Pascal-like behavioral VHDL subset and capabilities are developing for transforming the same VHDL subset into standard software development languages. However, the process of synthesizing behavioral VHDL from systems level requirements has not been addressed. Users are required to write behavioral VHDL descriptions of their components in a purely operational manner. This results in implementational bias and premature hardware/software allocation decisions. We propose automating this process by expressing systems level requirements in a declarative specification language and using standard software synthesis techniques to generate behavioral VHDL from them.

1 Introduction

The overall goal of this research is synthesis of composite computing systems using traditional software synthesis techniques. A composite computing system is defined as a collection of computation units that may be implemented either software or hardware components. To achieve this end, the high-level approach described in Figure 1.

The general flow of information through the system is as follows: (a) Design requirements (including constraints) are parsed to generate a decorated abstract syntax tree used by synthesis processes; (b) the problem may be decomposed into components; (c) an algorithm is synthesized for each component; and (d) The assemblages of components, the general algorithms and abstract syntax tree are transformed into an appropriate design representation. Given this design methodology, this research is decomposed into the following sub-goals:

1. Representation of system and component requirements.
2. Generation of an intermediate form to support synthesis
3. Synthesis of component designs
4. Generation of output in an appropriate design representation language

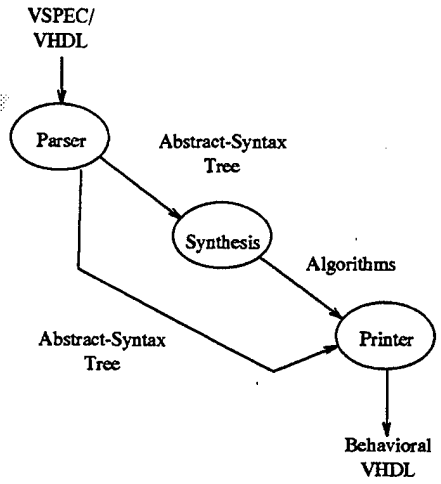


Figure 1: Flow of information through the synthesis process

This paper deals primarily with our specification language, called VSPEC, and the methods used to synthesize algorithms from requirements suitable for use in behavioral VHDL. VSPEC describes computation units axiomatically, specifying an input precondition and an output post condition.[1] VSPEC is a

[1] The process of parsing VSPEC to generate the appropriate

Larch [5] interface language for VHDL that translates both into the Larch Shared Language [5] and the high level programming language REFINE. The transformation of VSPEC into REFINE expresses the requirements in an independent form suitable for use by various software synthesis tools including KIDS [14] and BENTON [2].

1.1 Experimental Domain

Our current domain is rapid prototyping of digital signal processing systems. This work is directed towards automated synthesis of board- and MCM-level signal processors from systems level requirements. This synthesis domain includes ASICs, off-the-shelf components including CPUs, and embedded software.

The design representation language for this effort is mandated to be VHDL for hardware components and C for software components. In addition, all software components will be specified in VHDL first, then transformed into C as required by the sponsoring agency. Selection of VHDL is due to the domain's heterogeneous nature and the United States Department of Defense acceptance of VHDL as a systems representation language. Selection of C is due to the ready availability of C compilers for off-the-shelf digital signal processors and existing capabilities for performing VHDL to C transformations.

The general approach is synthesis of VHDL to represent both hardware and software components. Capabilities currently exist for transforming a rich subset of behavioral VHDL into RTL level VHDL suitable for synthesis and fabrication [11]. Capabilities also exist for transforming behavioral VHDL into compilable C code. Thus, we can achieve our objective by taking a requirements description of a system, transform the requirements description into behavioral VHDL and synthesize hardware and software components.

1.2 Axiomatic Specification

Specifying computation units using axiomatic specifications involves defining a transform by specifying an input precondition and an output postcondition. Given the input precondition holds, the transform must guarantee that the output postcondition is made true. Smith [13] suggests that such a specification be an algebra specifying the domain, range, input precondition and output postcondition. Thus, a function such as in Figure 2, may be described in terms of its domain (D), range (R), input precondition $I(x)$, and output postcondition, $O(x, z)$. When $I(x)$ holds for some input x of type D, the procedure must return some element z of type R such that $O(x, z)$ holds. A function $F(x) = z$ satisfies this specification when for any x such that $I(x)$ holds, $F(x)$ generates z such that $O(x, z)$ holds. Formally:

$$\forall x : D \bullet I(x) \wedge F(x) = z \Rightarrow \exists z : R \bullet O(x, z) \quad (1)$$

internal representation is a simple compiler problem. The process of generating VHDL source from REFINE [1] algorithms is a simple lateral transformation.

This work relies on the assumption that hardware components may be specified in the same manner. Specifically, that the transform associated with a hardware component can be defined by an appropriately selected domain, range, input precondition and output postcondition. A second assumption is that such axiomatic specifications can be used to synthesize hardware components. The first assumption, that hardware can be specified axiomatically, is easily made and is commonly used in formal verification of hardware. The second assumption is made based on the similarity between behavioral specification and traditional programming. The `process` component of VHDL supports specification of behavior using an Ada-like language. If requirements for Ada programs can be synthesized from requirements, then it stands to reason that VHDL programs can. Semantically, VHDL and Ada differ substantially - the bulk of this paper addresses some of those differences.

2 VSPEC

VSPEC is a Larch interface language [5] for VHDL. The VSPEC interface language annotates the VHDL `entity` structure adding component requirements in terms of precondition, postcondition, performance constraints and state. Each structure in the VSPEC interface language translates into a formal definition in a shared language. VSPEC differs from a typical Larch interface language in that the primary shared language is REFINE rather than the Larch Shared Language (The reasons for this difference will be discussed later). To understand the VSPEC language, one must first have a cursory understanding of how VHDL represents systems.

2.1 VHDL

VHDL [9] is a specification language for digital systems whose structure and appearance is similar to Ada [15]. Although this structural similarity exists, it is somewhat deceiving because the semantics of a VHDL specification differ substantially from a similarly structured Ada program.

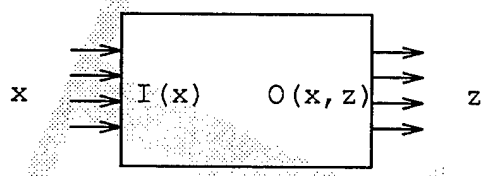
A system is described in VHDL by describing its constituent components and relationships between them. VHDL specifications consist of three fundamental construct types: (a) `entity` constructs describing component interfaces; (b) one or more `architecture` constructs describing each component's behavior or structure; and (c) `configuration` constructs associating `entities` with specific `architectures`. Thus, an `entity` represents an interface, several `architectures` represent behavior and structure, and a `configuration` indicates a specific architecture to represent the behavior of a component for a specific design task.

2.1.1 Entity Structures

An `entity` specifies the interface of each VHDL component much as an Ada public declaration specifies the interface of a procedure. The `entity` construct names the component and defines its `ports`. Ports are the hardware equivalent of parameters and represent the inputs and outputs, their types, and the

```
function F(x:D) : R
  begin
  I(x)
   -- Function Body
  O(x,z)
  return z
end;
```

a)



b)

Figure 2: Axiomatic descriptions of: (a) a typical procedure; and (b) a typical hardware component.

direction of data flow. VHDL **entity** structures are connected by connecting theory **ports**. Figure 3 is an **entity** describing a simple S-R latch. Note that the **entity** describes only the component interface, not its behavioral requirements or constraints.

```
entity sr_latch is
  port (s,r: in bit; q,qb: buffer bit);
end sr_latch;
```

Figure 3: A VHDL **entity** describing the interface to an S-R latch.

Parameters defined in the **port** definition are referred to as *signals*. Signals in VHDL are very similar to variables and parameters in a traditional programming language. Variables also exist in VHDL locally to processes, however in this work signal assignment is assumed to include variable assignment. When defining the behavior of a VHDL entity, relationships between input and output ports are defined, much as relationships between input and output parameters are defined in a traditional programming language.

2.1.2 Behavioral Specification

VHDL supports specification of a component's behavior directly using an operational description language, or indirectly using an assembly of other components. Behavioral specification involves writing a VHDL "program" in an operational VHDL subset similar in appearance to Ada. This subset includes familiar control structures and data types standard in procedural programming languages as well as signal assignment and synchronization constructs necessary to naturally specify hardware components. Figure 4 shows a behavioral description of the S-R latch.

The means of specifying behavior used in Figure 4 involves concurrent signal assignment statements. The values of **q** and **qb** are updated using the "**<=**" signal assignment operator. In this specification, value assignment to **q** and **qb** occurs simultaneously. Thus, the first assignment statement does not alter the program state prior to evaluating maintaining the original value of **q** for the second assignment statement. The VHDL code used to generate the assigned

```
architecture behavior of sr_latch is
begin
  q <= NOT (qb AND s);
  qb <= NOT (q AND r);
end behavior;
```

Figure 4: A VHDL **architecture** describing the behavior of an S-R latch.

value is a *driver*. There should exist one and only one driver for each output signal.

An alternative specification involves the use of **process** blocks. In a **process**, assignments do not occur simultaneously and statements execute in a manner similar to a traditional programming language. Thus, the VHDL fragment from Figure 5 requires the introduction of a temporary variable as is traditional in an imperative language. Note that the two behaviors specified using concurrent assignments and processes specify identical behaviors.

```
architecture behavior of sr_latch IS
begin
  p1: process
        variable tmp : bit;
      begin
        tmp := q;
        q <= not (qb and s);
        qb <= not (tmp and r);
    end process;
end behavior;
```

Figure 5: A VHDL **architecture** describing the behavior of an S-R latch using a single process.

If multiple processes exist in an architecture, all processes execute simultaneously. Thus, concurrent assignment statements described previously are a shorthand notation for a collection of processes with single assignments to output signals. The process equivalent of Figure 4 is shown in Figure 6

The parallels between **process** descriptions and

```
architecture behavior of sr_latch is
begin
  p1: process begin
        q <= NOT (qb AND s);
      end process;
  p2: process begin
        qb <= NOT (q AND r);
      end process;
end behavior;
```

Figure 6: A VHDL **architecture** describing the behavior of an S-R latch.

traditional programming languages are exploited to synthesize behaviors for single entities. The objective is synthesis of code for **process** statements and/or concurrent assignments. Problems are decomposed with respect to output ports and composed using the concurrent assignment or **process** facilities.

2.1.3 Structural Specification

Structural specification involves specifying a collection of VHDL **entity** components and connections between them. Using the **component** statement, the **architecture** specifies the components used in the assemblage and assigns local names to ports. The body of the structural architecture names each local component, assigns a component from the declarative section to it, and specifies connections involving the local component using local parameter names. Figure 7 shows a structural specification of an S-R latch.

```
architecture structure of sr_latch is
  component nor2
    port(a,b : in bit; c : out bit);
begin
  n1: nor2 port map (s,qb,q);
  n2: nor2 port map (r,q,qb);
end structure;
```

Figure 7: A VHDL **architecture** describing the structure of an S-R latch.

Together the **entity** and **architecture** constructs describe a component's inputs, functional behavior and structure. Many architectures may exist for a single **entity**, thus the **configuration** structure is used to specify what architecture should be associated with each entity. A typical VHDL design process involves specifying component interfaces, specifying behavior and refining the behavior to specify an implementation as a structural specification.

Given a behavioral description, there are automated and semi-automated means of refining that description. It is currently possible to synthesize directly implementable designs from behavioral VHDL as large as small CPUs [11]. Many commercial VHDL support environments include synthesis subsystems. Thus, prototype system synthesis is achieved by generating behavioral VHDL and using lower level synthesis tools to generate code, ASICs, and board layouts.

2.2 VSPEC Entities

VSPEC adds six declarative clauses to the VHDL **entity**: (1) the **state** clause declares variables representing the state of the component; (2) the **requires**[2] states the component's precondition and is a function mapping **entity** input and state variables onto the boolean set; (3) the **ensures** clause states the component's post condition and is a function mapping input, output and state variables onto the boolean set; (4) the **modifies** clause names input, output and state variables whose values may be changed by the component; (5) the **constrained by** clause states performance constraints associated with the component; and (6) the **based on** clause associates primitive and user defined types with shared language representations.

Each clause is stated as a logical expression (with the exception of the **modifies** and **based on** clauses) in typed first order predicate calculus with equality, extended to include set and sequence theories. The only variables allowable in the logical expressions are defined in the **entity**'s **port** definition, the VSPEC **state** clause, or defined locally in a logical expression as a quantified variable. All variables must be typed and typing requirements are checked by the VSPEC parser.

The VSPEC parser transforms each clause into a REFINE logical expression used to drive synthesis and analysis algorithms. Figure 8a shows a generic VSPEC **entity** definition with each VSPEC clause.

2.3 Representation of Architectures

VHDL represents connected collections of components using architectures as shown in Figure 7. Components are connected and their parameters used to indicate interconnection. The example shown in Figure 7 defines a two stage, batch sequential approach to searching a collection of values. The input list is sorted and a binary search is applied to the result of sorting. The **architecture** represents the batch sequential approach by defining a sorting component, defining a binary searching component, and connecting the outputs of the sorter to the inputs of the searcher. Note however that the search and sort component's implementation details are not specified. Specific algorithms must be synthesized at some later point.

Thus, the **architecture** notion is used in conjunction with VHDL **entity** components with VSPEC annotation to represent system architectures. Representation of requirements for multi-component systems also allows VSPEC to represent composite, multi-component systems by supporting specification of hardware executing software processes and complex device intercommunication.

[2] In earlier versions of VSPEC and earlier papers, the requires clause case called the assumes clause

```
entity example is
    port (a,b: in bit; c: out bit);
  modifies c;
  state s: bit;
  requires I(a,b);
  ensures O(a,b,s,c) and D(a,b,s,s');
  constrained by Q;
end example;
```

a)

```
D = bit×bit× bit
R = bit× bit
I = R(a,b,s)
O = O(a,b,s,c) ∧ D(a,b,s,s')
C = Q
```

b)

Figure 8: VSPEC definition and associated tuple representation.

```
entity search is
  port (input: buffer array of integer;
        key: in integer;
        value: out integer);
  modifies value;
  ensures
    ∀ x : integer value = x ⇒ x ∈ input;
end example;

architecture bat-seq of search is
  component sorter
    port (input: in array of integer;
          output: out array of integer);
  component bin_search
    port (input: in array of integer;
          key: in integer;
          value: out integer);
begin
  b1: c1 port map(x,y);
  b2: c2 port map(y,z);
end;
```

```
entity sort is
  port (input: in array of integer;
        output: out array of integer);
  modifies output;
  ensures bag(input) = bag(output) ∧
          sorted(output
end sort;

entity bin_search is
  port (input: buffer array of integer;
        key: in integer;
        value: out integer);
  modifies out;
  requires sorted(input);
    ∀ x : integer value = x ⇒ x ∈ input;
end bin_search;
```

Figure 9: Using a VHDL architecture to represent general structures. Note that a VSPEC entity is used to represent the requirements of each component

3 Parsing VSPEC

Algorithm synthesis does not operate on raw VSPEC. Before algorithm synthesis begins, the VSPEC definition is parsed into a decorated abstract syntax tree. The abstract syntax tree is represented using the REFINE object-base and the parser written in the DIALECT system.

In the abstract syntax tree, each **entity** is represented by its constituent components from the interface language. Namely, the **port** and **state** clauses representing the system interface and internal state, the input precondition, the output post condition, and any existing performance constraints. Together, these define a specification as a problem theory [13] supporting use of KIDS and other similar transform systems.

The abstract syntax tree also represents **architectures**. Each **architecture** is linked to the **entity** representing its interface and requirements. Refining our synthesis objective leads to the goal of associating each **entity** with at least one **architecture** with a behavior description, or a structural description whose components have complete behavioral descriptions at some level of abstraction. Figure 8b shows the result of parsing a VSPEC **entity**.

4 VHDL Synthesis From VSPEC

After VSPEC is parsed into the abstract syntax tree form, synthesis activities begin. For each **entity**, a suitable **architecture** must be synthesized. From the **port** descriptions and VSPEC clauses, a domain theory is formed and represented using the DRIO notation proposed by Smith [12, 13, 14]. The user guides the selection of a general structure involving either a single, behavioral architecture, or structural architecture specifying a configuration of components.

4.1 Synthesis Goals

Although VHDL and Ada share structural similarities, VHDL should not be viewed as simply a programming language. Several characteristics of VHDL representations must be accounted for in the synthesis process. These include the co-existence of **entities**, the state machine nature of entity descriptions, signal attributes, and concurrent assignment.

A naive examination of VHDL may lead to the belief that the **entity** component is equivalent to a procedure or function in a traditional imperative language. Thus, connections between components define a sequential control flow. In a VHDL description, **entity** components represent concurrently existing devices and processes. Activation of components occurs due to parameter changes, not due to explicit calls and parameter passing. Each **entity** description has a sensitivity list indicating what parameter changes can cause its invocation. When invoked, the **architecture** implementing the entity is executed to completion, interacting with other **entity** structures only through changing port values and **wait** statements. Although an **entity** is the basic computing element in VHDL as a procedure is in Ada, an **entity**'s behavior more closely represents a process than a procedure.

The general goal of a VHDL synthesis activity driven by a VSPEC specification is to synthesize a function, $F(x)$, such that:

$$\forall x : D \bullet I(x) \Rightarrow \exists z : R \bullet O(x, z) \wedge F(x) = z \qquad (2)$$

where:

- D is the cartesian product of sorts associated with **in**, **inout** and **buffer** parameters.
- R is the cartesian product of sorts associated with **out**, **inout** parameters and only those **buffer** parameters named in the **modifies** clause.
- $I(x)$ is the input precondition defined in the **requires** clause.
- $O(x, z)$ is the output postcondition defined in the **ensures** clause.

4.2 State Based Solutions

VHDL reflects the common view of hardware components as state machines. Unlike a typical subprogram, an **entity**'s local storage is not initialized for each invocation - local variables and some parameters maintain their previous values. Thus, values of locally defined signals and variables, and ports define the state of the component. In Figure 4, the previous values of **q** and **qb** are used to generate the next values.

The structure of a traditional axiomatic specification from Section 1.2 is defined over the inputs and outputs of the specified component. No mention is made of the internal state of the component. The brute force approach would be altering the **entity** definition to include state variables as inputs.

State-based system synthesis is achieved by synthesizing a transform that includes anything maintaining its state from one invocation to the next as a part of both the domain and range of the transform. Consider the VSPEC example from Figure 8. To satisfy this VSPEC specification, we must synthesize one or more transforms that collectively satisfy:

```
D = bit×bit×bit
R = bit×bit
I(x:D) = R(a,b,s)
O(x:D,z:R) = O(<a,b,s>,x) ∧ D(<a,b,s>,s)
```

The domain and range of the **entity** being synthesized are different than the domain and range of the synthesis goal. The **entity** domain and range are both augmented to include types of state variables. Thus, the synthesized function will produce values for **entity** output signals and signals and variables maintaining state. The state variables of a VSPEC entity include and variables defined in the **state** clause and ports defined as type **buffer**, **out**, or **inout**. The synthesis goal stated earlier is modified such that the domain and range include values of state variables. Appropriate **buffer** signals have already been included in the original domain.

Note the single function produces state transition and output. Thus, the resulting system is either a Moore or Mealy type machine depending on the signals and variables involved in calculating outputs.

4.3 Managing Concurrency

Behavioral VHDL heavily utilizes concurrent signal assignment and concurrent processes. Viewed as a sequential program, the specification from Figure 4 is incorrect. The contents of q would be updated before qb replacing the previous q needed for qb. This is an example of the classic value swap problem in traditional programming languages. In behavioral VHDL, these assignments occur concurrently. When assignments occur concurrently, the specification will function correctly.

Concurrent assignments and concurrent processes begin from the same state and cause state changes at the same time. Thus, any problem may be decomposed into processes that generate outputs for a subset of output signals. If no output signal is driven by multiple assignments and processes do not interact via `wait` statements or shared local variables, the composition of those processes is trivial. Full advantage of the independence of processes and assignments is taken when partitioning problems.

4.4 Partitioning

The brute force synthesis approach is to generate an algorithm that accepts an element from D and generates an appropriate element of R. This function is translated into a single VHDL process that executes and updates all output signals. Figure 10 illustrates such a transform.

A more appropriate synthesis method takes advantage of the VHDL process and concurrent signal assignment concepts. The requirements of each function being synthesized is decomposed into requirements for each signal, or requirements for disjoint subsets of signals defined in the function's range. When evaluating concurrent signal assignments, each signal driver is evaluated independently *from the same initial state* and results are concurrently assigned to outputs. Thus, the evaluation of each driver has no effect on other drivers.

When synthesizing functions for drivers, full advantage is taken of this independence. The synthesis obligation is decomposed into several simpler obligations for subsets of the output signals. These functions are composed as processes in an architecture. The composition will be correct if: (a) each output signal appears on the left side of an assignment in only one driver; (b) the conjunction of postconditions from each driver satisfies the overall postcondition; and (c) satisfying the input condition implies the input condition of each driver is satisfied.

Formally, synthesizing algorithms for collections of signals involves generating the set of functions $f_1, f_2, \ldots, f_n$ where $D_k, R_k, I_k(x), O_k(x,z)$ and C_k describe the domain, range, input condition, output condition and constraints of $f_k(x)$. The following two conditions must also hold:

$$I(x) \Rightarrow \bigwedge_{k=1}^{n} I_k(x_k) \tag{3}$$

$$\bigwedge_{k=1}^{n} O_k(x_k, z_k) \Rightarrow O(x,z) \tag{4}$$

Equation 3 assures that if the overall input precondition is met, individual driver preconditions are also met. If this were not the case, then it would be possible for a function to fail when the precondition of the overall entity is met. Equation 4 assures that if each driver postcondition is satisfied, the overall output condition is satisfied. If this were not the case, then the collection of synthesized functions will not necessarily generate all necessary output values.

If the mapping from each output or state variable to the function that generates it is injective, then a driver is synthesized for each output. Thus, concurrent signal assignments are used to assemble the drivers into a single component. Otherwise, a `process` and necessary local storage are created for each driver. Both options are shown in Figure 11.

It should be noted that each of the drivers synthesized functions independently. From a synthesis perspective, this eliminates the need to verify that no harmful interactions occur between drivers. However, a system is rarely developed as a collection of independent components. To synthesize realistic VHDL systems, multi-component systems with realistic degrees of interaction must be synthesized.

4.5 Signal Attributes

Software systems deal primarily with stable variable values. Hardware representation systems must represent not only signal values, but how those values change. Consider a device with a leading-edge driven clock. If the clock is viewed as a binary value, only two states can be represented. VHDL provides function attributes of the form *sym'att* where *sym* is a defined symbol name and *att* is an attribute defined for that symbol. Attributes such as `delayed`, `stable`, `quiet`, and `transaction` are defined for all symbols representing signals. For example, the `event` attribute returns a true value if its associated signal just changed values. Thus, the following VHDL statement represents the conditional for an event that should occur on the rising edge of signal `clk`:

```
clk='1' and clk'event
```

Managing signal attributes appears to be a difficult problem. However, defining predicates and theories for needed attributes supports their inclusion in the specification process. The `clk'event` attribute reference can easily be represented as the predicate `event(clk)`. Furthermore, adjusting the syntax of the interface language allows specification of the attribute using the VHDL form. Figure 12 shows an adaptation of the SR latch specification to include a clock signal and form an edge triggered flip-flop.

Although some VHDL characteristics may not feel natural to a traditional programmer, they are quite

```
entity sr_latch is
  port (s,r : in bit,
        q,qb: buffer bit)
  ensures
    q' = ~(s and qb) and
    qb' = ~(r and q)
end sr_latch;

function sr_latch(s,r,q,qb: bit)
    : tuple(bit,bit)
  <not(s and qb), not(r and q)>
```

```
architecture mono of sr_latch is

  procedure sr_char(s,r,q,qb: in bit;
                    nq,nqb: out bit) is
  begin
    nq := not(qb and s);
    nqb := not(q and r);
  end sr_char;

begin
  b1: process
        variable nq, nqb: bit;
      begin
        sr_char(s,r,q,qb,nq,nqb);
        q <= nq;
        qb <= nqb;
      end process;
end mono;
```

Figure 10: Monolithic algorithm synthesis for a single VHDL entity.

```
architecture proc of test is
  b1: process
        variable tmp1,tmp2...tmpk : R1
    f1(x1,tmp1,tmp2...,tmpk);
    z1 <= tmp1;
    z2 <= tmp2;
    ...
    zk <= tmpk
  end process;
  b2: process
        variable tmpk+1,tmpk+2..tmpj : R2
    f2(x2,tmpk+1,tmpk+2...,tmpj);
    zk+1 <= tmpk+1;
    zk+2 <= tmpk+2;
    ...
    zj <= tmpj
  end process;
  ...
    bm: process
        variable tmpn-1,tmpn : Rm
    fm(xm,tmpn-1,tmpn);
    zn-1 <= tmpn-1;
    zn <= tmpn;
  end process;
end proc;
```

```
architecture concur of test is
  z1 <= f1(x1);
  z2 <= f2(x2);
  ...
  zn <= fn(xn);
end concur;
```

Figure 11: Assembling multiple algorithms into a single component using processes and concurrent signal assignment.

```
entity re_sr_latch is
  port (s,r : in bit,
        q,qb: buffer bit,
        clk: in bit)
  ensures
     (clk=1 and event(clk) and
      q' = ~(s and qb) and
      qb' = ~(r and q))
    or
     (q'=q and qb'=qb)
end re_sr_latch;
```

Figure 12: SR latch specification modified to specify a rising edge triggered SR flip-flop.

natural to hardware designers. In addition, each can be specified using traditional, axiomatic specification techniques. However, any successful VSPEC-related tool must be used by hardware designers. Thus, specification of this type of characteristic in VSPEC and synthesis of VHDL supporting these characteristics must be supported by VSPEC. This is the primary reason for using a Larch interface language - the designer works in a language supporting traditional hardware specification techniques and familiar constructs that have a formal interpretation.

5 Synthesis Techniques

Given the result of parsing a VSPEC entity, one may employ any number of synthesis techniques to derive an algorithm for the transform. In this work, formal synthesis techniques are employed. Specifically, a case-based reasoning approach based on the CYPRESS [12] operator_match problem solving technique and interfacing with KIDS [14].

5.1 Algorithm Synthesis

5.1.1 Direct Transformation

The simplest algorithm specification technique available is direct transformation. It should be used when the structure of the specification directly reflects the structure of VHDL used to implement it. The specification is transformed using simple syntactic techniques to generate VHDL code. This technique is similar to the specification to code option available in KIDS and other automatic programming systems.

5.1.2 Case-Based Reasoning

Case-based reasoning [10] uses similarities between problems to select solutions. The assumption is that similarity between problems implies similarity between solutions. In the BENTON system, case-based reasoning is used to retrieve and reuse specifications [2]. The same techniques are used to retrieve and reuse VHDL fragments described by VSPEC specifications. VSPEC is used to generate features from both the problem and potential solutions for similarity calculation. Case-based reasoning is useful primarily for retrieving potential solutions, but does not guarantee validity of the solution. Thus, after the solution is retrieved and adapted, a further proof obligation exists. VSPEC makes this obligation simpler and supports means for correctness preserving adaptation using derived antecedent, but it does not avoid the obligation.

5.1.3 Formal Transformation

The chief algorithm synthesis tool is KIDS [14, 13]. KIDS is based on the formal composition of an algorithm theory representing a problem solving methodology and a domain theory representing the problem itself. The DRIO specification generated by the VSPEC parser is motivated chiefly by specification format used by KIDS, however other synthesis systems frequently use similar means for representing specifications [8].

In order to generate algorithms using KIDS, one must specify a complete domain theory, of which the specification itself is only a part. One must also use REFINE to specify laws and auxiliary functions that define the transform itself. The transformation from interface language to REFINE accomplishes some of this along with libraries of general theories describing operators over types. In general, specifications beyond those directly specified by VSPEC are required for the synthesis process to complete effectively.

5.2 Architecture Synthesis

The most active area of this research is synthesis of multi-component systems. Given a high level VSPEC specification, generate a system involving a collection of interconnected `entity`'s rather than simply a collection of independent processes.

5.2.1 Case-Based Reasoning

The simplest technique currently used is applying case-based reasoning to retrieve and adapt multi-process architectures. Architectures take the form of procedural networks with each action representing a single component. In a typical procedural network, an action is represented by a precondition and post-condition, thus the representation adapts naturally to specification of some architectures. Actions are specialized using algorithmic synthesis techniques, antecedent derivation, or heuristic adaptation. As before, the results of some adaptation processes require that the resultant algorithm be verified. If the architecture is known to be correct and is specialized using correctness preserving operations, verification is typically not required.

5.2.2 Formal Synthesis

General architectures can be synthesized using the KIDS approach by developing algorithm theories to support architecture synthesis and by using antecedent derivation to discover missing components [3]. The batch sequential architecture for the search `entity` shown in Figure 9 can be synthesized

by selecting the binary search algorithm and using its precondition to derive the sort ENTITY.

The binary search algorithm takes a key value and a list of elements and returns the value discovered in the list. The precondition of binary search is that the input list must be ordered. Other preconditions may also be derived to fit this algorithm to the problem. There is no precondition associated with the search **entity** driving the synthesis process, thus there is no assurance that the collection of inputs will be in order. Thus, a component must be developed to prepare the original input for use by the binary search routine. This technique is very similar to techniques used by CYPRESS and **kids** to synthesize divide-and-conquer algorithms [12]. The specification of this new component will be as follows:

- $D = D_s$
- $R = D_{bs}$
- $I(x) = I_s(x)$
- $O(x, z) = I_{bs}(z)$

where $D_s, R_s, \ldots$ are associated with the original search specification and $D_{bs}, R_{bs}, \ldots$ are associated with the binary search specification. The resultant specification is almost the sorting specification with no precondition and a sorted output condition. Note the missing **bag(x)=bag(z)** element in the generated specification.

Arbitrarily complex sequences of **entity**'s may be specified in this manner by: (a) repeating the batch sequential process for discovered components; and (b) generating similar techniques for batch parallel and conditional branching. Note that a control strategy is not proposed here. The user must make control decisions at each synthesis stage. Thus, problems associated with some planning algorithms can be avoided.

5.2.3 Non-Sequential Architectures

The antecedent derivation techniques can effectively generate architectures where a clear order of execution exists and components do not engage in bidirectional communication.[3] Consider specification of a pair of transceivers or synchronously communicating devices. In order to synthesize such system using KIDS techniques, general algorithm theories must be developed describing various architectures. Antecedent derivation is useful even in these situations, but discovering missing components should eventually give way to specializing known architectures to specific problems.

6 Related Work

The approach taken in constructing the VSPEC language is based heavily on the Larch/Modula-3 interface language [7]. Another parallel effort in developing a Larch interface language is underway at Odyssey Research Associates [6]. This interface language uses the Larch Shared Language rather than REFINE and is targeted towards formal VHDL verification, not synthesis. The techniques used in this work are being extended from Penelope [4], an Ada verification system. VSPEC could potentially support verification, however its prime motivation is driving synthesis processes.

Most of the synthesis aspects of this work and the specification of components in terms of domain, range, input precondition and output postcondition is based on application of algorithm theories to program synthesis. These techniques were proposed by Smith [13] and implemented in the CYPRESS [12] and KIDS [14] systems.

7 Future Directions

Three directions currently dominate this research effort: (1) development of KIDS algorithm theories for general architectures; (2) management of constraints during the design activity; and (3) migration of the general technique away from VHDL.

An algorithm theory represents a general problem solving technique. Using a multi-component architecture is one such general technique, however no theory exists for its application in the current KIDS system. To extend these techniques to larger systems, general algorithm theories must be developed. Proposed techniques for batch sequential systems are shown here and similar techniques are proposed for batch parallel. However, more complex architectures must be developed, particularly for communicating systems.

Currently VSPEC represents several types of constraints. At each stage in the design process, these constraints can be checked in the abstract syntax tree. Thus, constraint violations can be detected. Of particular difficulty is management of propagation time. Odyssey Research Associates [6] takes the approach of associating events with time points in the interface language. This requires using a temporal logic in the verification activity. VSPEC uses an interval representation to define the time from input signal(s) arrival to output signal(s) generation. This separates timing issues from the functional specification. Although timing constraints can be verified, they must be included in the actual synthesis process eventually.

Finally, the Generic Abstract Syntax Tree (GAST) is being developed to serve as a general representation for systems requirements. The objective is to either adapt VSPEC to new source languages or use existing Larch interface languages to generate GAST requirement representations. A parser is written to generate GAST from each language and synthesis (and potentially analysis) tasks performed on the GAST representation. The resulting algorithms plus the GAST representation are transformed into the output language of choice. Note that both the input parsing and output transformation are purely syntactic activities, thus existing technologies can be used to construct these components. Using taking this approach, VSPEC techniques may be more generally inserted in the systems development and prototyping process.

[3] It has not been determined that antecedent derivation cannot be used for such situations. It simply has not been demonstrated that it can.

8 Acknowledgment

Support for this work was provided in part by the Advanced Research Projects Agency and monitored by Wright Labs under the RASSP Technology Program, contract number F33615-93-C-1316. The authors wish to thank Wright Labs and ARPA for their continuing support.

The authors wish to thank Dr. Paul Bailor, Mark Gerken and Frank Young of the Air Force Institute of Technology for their help and comments on this research. In particular we wish to acknowledge their contributions in architecture synthesis and the batch sequential algorithm synthesis technique.

References

[1] L. Abraido-Fandino. An overview of refine 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering*, Madrid, Spain, April 1987.

[2] P. Alexander. Combining transformational and derivational analogy in Larch specification generation. In *Proceedings of The 6th International Conference on Software Engineering and Knowledge Engineering*, pages 131–138, Riga, Latvia, June 1995. Knwoledge Systems Institute.

[3] P. Bailor, M. Gerken, and F. Young. Personal communication, 1994. (technical report pending).

[4] D. Guaspari. Penelope: An Ada Verification System. In *Proceedingsof Tri-Ada '89*, pages 216–224, Pittsburgh, PA, October 1989.

[5] J. Guttag and J. Horning. *Larch: Languages and tools for formal specification.* Texts and Monographs in Computer Science. Springer-Verlag, New York, NY, 1993.

[6] D. Jamsek and M. Bickford. Formal Verification of VHDL Models. Technical Report RL-TR-94-3, Rome Laboratory, Griffiss Air Force Base, NY, March 1994.

[7] K. Jones. LM3: A Larch Interface Language for Modula-3. Technical Report 72, DEC Systems Research Center, Palo Alto, CA, 1991.

[8] Z. Manna and R. Waldinger. A Deductive Approach to Algorithm Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[9] D. Perry. *VHDL.* McGraw-Hill, New York, NY, 1st edition, 1991.

[10] C. Riesbeck and R. Schank. *Inside Case-Based Reasoning.* Lawrence Earlbaum Associates, Hillsdale, NJ, 1989.

[11] Jayanta Roy, Nand Kumar, Rajiv Dutta, and Ranga Vemuri. DSS: A Distributed High-Level Synthesis System. *IEEE Design&Test of Computers*, pages 18–32, June 1992.

[12] D. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, Sept. 1985.

[13] D. Smith. Algorithm Theories and Design Tactics. *Science of Computer Programming*, 14:305–321, 1990.

[14] D. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, Sept. 1990.

[15] United States Department of Defense, Washington, DC. *Reference Manual for the Ada Programming Language*, 1st edition, Feburary 1983.