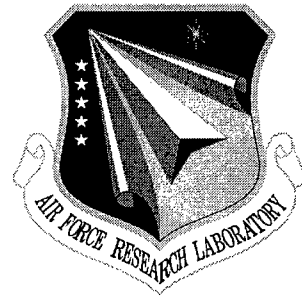


AFRL-IF-RS-TR-1999-123

Final Technical Report

June 1999



AUTOMATIC SYNTHESIS OF PLANNERS AND SCHEDULERS

Kestrel Institute

Douglas R. Smith, Junbo Liu, and Stephen J. Westfold

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

19990817 099


**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

DTIC QUALITY INSPECTED 4

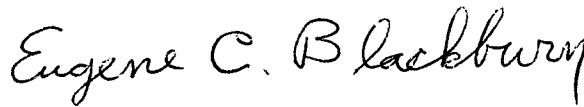
This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1999-123 has been reviewed and is approved for publication.

APPROVED:


JOHN F. LEMMER
Project Engineer

FOR THE DIRECTOR:


EUGENE C. BLACKBURN, Chief
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

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 June 1999	3. REPORT TYPE AND DATES COVERED Final Sep 95 - Sep 98		
4. TITLE AND SUBTITLE AUTOMATIC SYNTHESIS OF PLANNERS AND SCHEDULERS		5. FUNDING NUMBERS C - F30602-95-C-0247 PE - 63226E PR - C677 TA - 01 Wu - 17		
6. AUTHOR(S) Douglas R. Smith, Junbo Liu, and Stephen J. Westfold		8. PERFORMING ORGANIZATION REPORT NUMBER N/A		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Kestrel Institute 3260 Hillview Avenue Palo Alto CA 94304		10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-1999-123		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/IFTB 525 Brooks Road Rome NY 13441-4505		11. SUPPLEMENTARY NOTES AFRL Project Engineer: John Lemmer/IFTB/(315) 330-3657		
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.		12b. DISTRIBUTION CODE		
13. ABSTRACT (Maximum 200 words) This report describes our research on the transformational development of transportation plans and schedules. Our approach to developing scheduling software involves several stages. The first step is to develop a formal model of the transportation scheduling domain, called a domain theory. Second, the constraints, objectives, and preferences of a particular scheduling problem are stated within a domain theory as a problem specification. Finally, an executable scheduler is produced semi-automatically by applying a sequence of transformations to the problem specification. The transformations embody programming knowledge about algorithms, data structures, program optimization techniques, etc. The result of the transformation process is executable code that is consistent with the given problem specification. Furthermore, the resulting code can be extremely efficient.				
14. SUBJECT TERMS Campaign Scheduling, Domaine Theory, Campaign Planning			15. NUMBER OF PAGES 32	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

1. Executive Summary

This report describes our research on transportation planning and scheduling supported by the ARPA/Rome Lab Planning Initiative (ARPI). The main goal of this project was to develop generic tools to support the construction of flexible, high-performance planning and scheduling software. Our technical approach is based on *specification refinement* technology which allows the systematic machine-supported development of software from requirement specifications. The development process can produce highly efficient code along with a proof of the code's correctness.

Our approach to developing scheduling software involves several stages. The first step is to develop a formal model of the transportation scheduling domain, called a *domain theory*. Second, the constraints, objectives, and preferences of a particular scheduling problem are stated within a domain theory as a *problem specification*. Finally, an executable scheduler is produced by applying a sequence of *refinements* to the problem specification. The refinements embody programming knowledge about algorithms, data structures, program optimization techniques, etc. The result of the refinement process is executable code that is consistent with the problem specification.

Under this contract we developed a system called Planware, a domain-specific generator of high-performance scheduling software. Architecturally, Planware is an extension of the Specware system with domain-independent and domain-dependent parts. The domain-independent part includes a general algorithm design facility (including mechanisms to synthesize global-search and constraint propagation algorithms) and support for constructive theorem-proving. The domain-dependent part includes scheduling domain knowledge and architecture representations, and other domain-specific refinement knowledge that relates the scheduling domain to general algorithm design and data type refinement. Using Planware, the user interactively specifies a problem and then the system automatically generates a formal specification and refines it.

Another aspect of our project involved applying the Planware methodology to support automated synthesis of planners customized to specific domains. This work was carried out at Arizona State University under a sub-contract from Kestrel. In the approach we developed, a domain expert is allowed to input domain-specific control knowledge, which is then folded into a pre-existing declarative theory of refinement planning, to generate planner code specific to that domain. We used this approach to successfully synthesize planners in several benchmark domains, such as logistics planning.

During this contract we also worked on various applications of our synthesis technology. We continued the development of ITAS (In-Theater Airlift Scheduler) for PACAF (Pacific Air Force) at Hickham AFB. The most recent version simultaneously schedules the following classes of resources: (1) aircraft, (2) aircrews and their duty day cycles, (3) ground crews for unloading, and (4) ramp space at ports. Together with BBN and Carnegie-Mellon University, we developed an advanced scheduling system for AMC TACC (Air Mobility Command, Tanker-Airlift Operations Center) at Scott AFB. This system is scheduled for integration by Logicon Corporation into the CAMPS system and deployed in 1999.

Contents

1. Executive Summary	i
2. Introduction	1
3. Planware	3
4. Specware	5
5. Planware design process	7
5.1. Abstract scheduling problems	8
5.1.1. Abstract scheduling specification	8
5.1.2. Refining to a particular scheduling problem	9
5.1.3. Example – transportation scheduling	11
5.2. Data-type reformulation	12
5.2.1. Example – transportation scheduling	13
5.3. Algorithm design	13
5.3.1. Example – transportation scheduling	16
5.4. Expression optimization	16
5.5. Automatic code generation	16
6. Synthesis of customized planners	17
7. Concluding remarks and future work	19
REFERENCES	20

List of Figures

1	Advanced Development Environment for Planning and Scheduling Software .	2
2	Partial-Order SPEC in Specware	5
3	DRO-SPEC in Specware	5
4	Specification Morphisms	6
5	Interpretation in Specware	6
6	Scheduling System Architecture	9
7	Scheduling Specifications in Specware	10
8	Due-Date Constraint	11
9	Resource Taxonomy	12
10	Task Taxonomy	12
11	Algorithm Design Diagram	14
12	Ladder Construction	15
13	domain-specific algorithm	16
14	Architectural overview of planner synthesis with KIDS in the CLAY approach. The theories of refinement planning and domain knowledge are declaratively specified to KIDS which in turn combines them to produce a customized planner for the domain. The resulting planner, like conventional planners, can handle any planning problem from the domain.	18
15	The CLAY architecture for writing planning theory. Each level answers a question relevant to that level of planning detail. CLAY uses KIDS' feature of theory import to modularize the domain-specific planning theory	18

2. Introduction

This report describes our research on the transformational development of transportation plans and schedules. Our approach to developing scheduling software involves several stages. The first step is to develop a formal model of the transportation scheduling domain, called a *domain theory*. Second, the constraints, objectives, and preferences of a particular scheduling problem are stated within a domain theory as a *problem specification*. Finally, an executable scheduler is produced semi-automatically by applying a sequence of *transformations* to the problem specification. The transformations embody programming knowledge about algorithms, data structures, program optimization techniques, etc. The result of the transformation process is executable code that is consistent with the given problem specification. Furthermore, the resulting code can be extremely efficient.

Transportation scheduling tools currently used by the U.S. government are based on models of the transportation domain that few people understand [7]. Consequently, users often do not trust that the scheduling results reflect the characteristics of the current situation. Our approach tries to address this issue by making the domain model and scheduling problem explicit and clear. If a scheduling situation arises which is not treated by existing scheduling tools, the user can specify the problem and generate a situation-specific scheduler.

One of the benefits of a transformational approach to scheduling is the synthesis of specialized constraint management code. Previous systems for performing scheduling in AI (e.g. [4, 3, 13, 12]) and Operations Research [1, 5] use constraint representations and operations that are geared for a broad class of problems, such as constraint satisfaction problems or linear programs. In contrast, transformational techniques can derive specialized representations for constraints and related data, and also derive efficient specialized code for constraint checking and constraint propagation.

Figure 1 describes our vision of an advanced environment for producing planning/scheduling software. Briefly, the idea is to rapidly develop a situation-specific domain model and problem specification using a knowledge-elicitation system, and then to *synthesize* high-performance planning and scheduling tools that are specialized to the current situation. The majority of users' interaction would be codifying the *domain theory* and specification of the current situation, to aid in synthesizing a customized planning/scheduling tool.

We now step through the process in more detail. Several classes of users are involved in the construction and use of a scheduling system.

One class of users, who include domain experts and specialists in model construction, interact with a knowledge elicitation system to help classify the features of the situation and select, compose, extend, and refine, (possibly abstract) models from a preexisting library of domain models. The result is a model and problem specification tailored to the details of the current situation (as closely as expertise and time permit).

Another class of users, who specialize in software design and formal modeling of programming knowledge, interact with a planning/scheduling synthesis system to develop code from the problem specification. The interaction involves composing components from a library of reusable parts, or selecting and applying representations of abstract programming knowledge

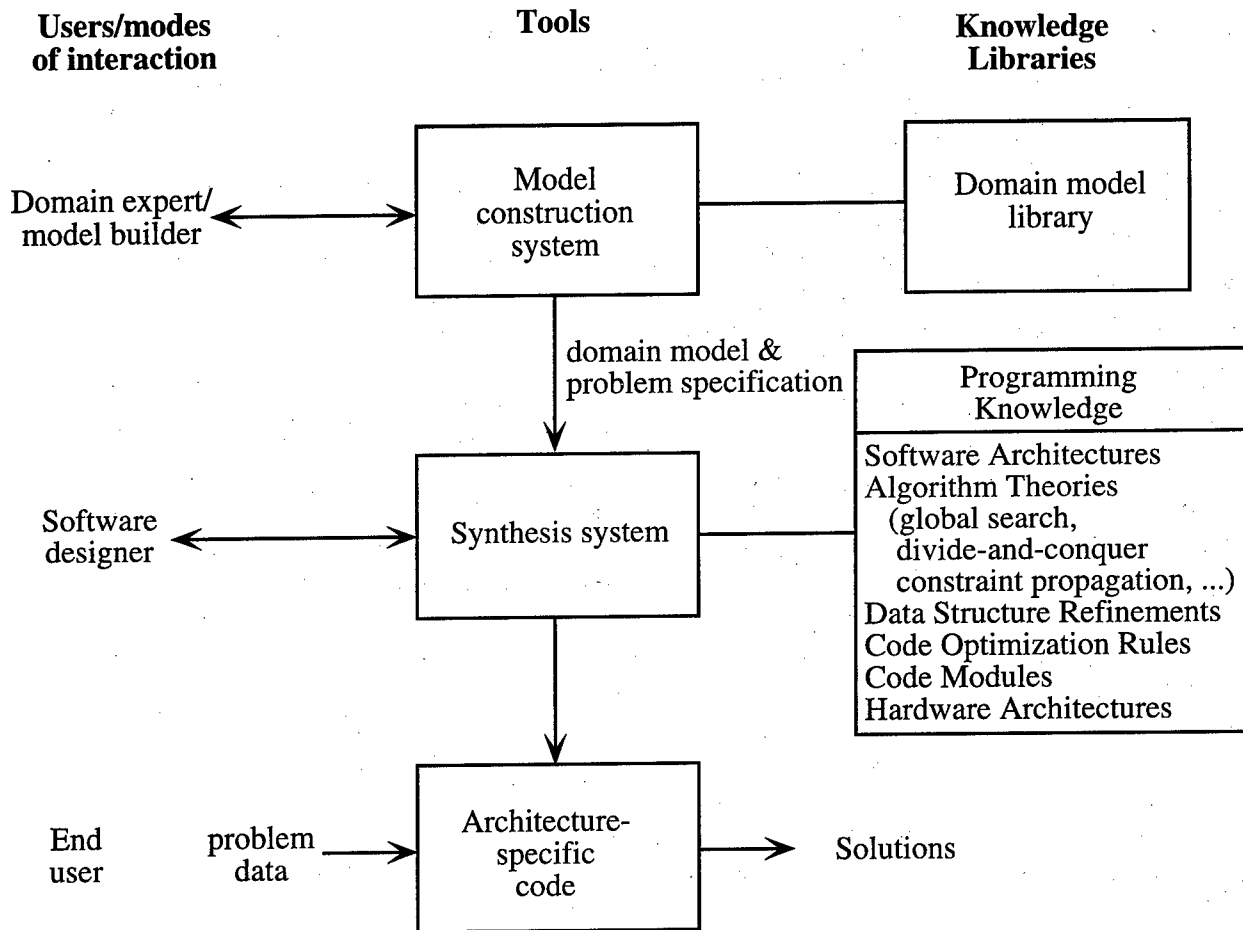


Figure 1: Advanced Development Environment for Planning and Scheduling Software

about algorithms, data structures, code optimization techniques, planning and scheduling-specific design strategies, and so on. Most of the design process is automated, with only a few high-level design decisions made by the developer. Another interactive task for this user is the evolution of the programming knowledge-base itself.

The output of the synthesis system is executable planning/scheduling code which can then be embedded in a planning/scheduling system and executed by an end-user.

There is a feedback loop implicit in the figure. The end user/domain expert using the synthesized code may detect missing constraints, or have preferences or other information not accounted for in the code. This information is fed back to the model-building stage and the process iterates. The fact that each synthesis step preserves consistency between problem specification and generated code means that *maintenance and evolution back up to the specification/domain model level*, not the code-level as in current practice.

We developed an approximation to this vision in the current project, called Planware, and demonstrated its feasibility. Planware is a domain-specific generator of high-performance scheduling software, currently being developed at Kestrel Institute. Architecturally, Planware is an extension of the Specware system with domain-independent and domain-dependent parts. The domain-independent part includes a general algorithm design facility (including mechanisms to synthesize global-search and constraint propagation algorithms), as well as support for theorem-proving and witness finding. The domain-dependent part includes scheduling domain knowledge and architecture representations, and other domain-specific refinement knowledge that relates the scheduling domain to general algorithm design and data type refinement. Using Planware, the user interactively specifies a problem and then the system automatically generates a formal specification and refines it.

We first present an overview of Planware, then review Specware foundations, and then go into detail regarding the theory of scheduling in Planware. We also provide some details on the sub-contract research focused on the synthesis of domain-specific planners. The Appendices describes foundational work on Designware which supports Planware, and describes our synthesis process for the Mission Planner in CAMPS.

3. Planware

This paper presents an overview of Planware, a generator of high-performance scheduling algorithms, currently being developed at Kestrel Institute. Our aim is to convey a sense of the rationale for Planware, the design process that it supports, the architecture of the current Planware system, and our results to date. The reader may find more detail in the references.

Architecturally, Planware is an extension of the Specware system [14], a system for developing formal specifications and refinements based on concepts from higher-order logic and category theory. Planware and Specware embody theoretical developments stemming from Kestrel's experience with previous systems, such as KIDS [9] and DTRE [2].

The goal of Planware is to allow experts in planning and scheduling to assemble quickly a specification of a scheduling problem, and to generate automatically a high-performance scheduler from it. The user's interactions with the system are designed to be entirely in the scheduling domain – the user does not need to read or write formal specifications, nor to understand the logical and category-theoretic foundations of the system. We have invested substantial effort in automating the construction of scheduling domain theories.

To assemble a requirement specification and underlying domain theory, Planware requires very little information from the user:

- to select from a menu various attributes that specify the tasks that need to be scheduled, and
- to select from a taxonomy of resource theories the particular kind of resource against which to schedule the tasks.

From this minimal amount of information, Planware can automatically

- generate a formal specification of the scheduling problem (plus the relevant background concepts that comprise a domain theory),
- reformulate the specification using datatype refinements to build some of the problem constraints directly into the schedule datatype, allowing a dramatic simplification of the specification,
- apply domain-independent knowledge about designing global search (backtracking) algorithms with constraint propagation,
- apply datatype refinements and optimization techniques, and finally
- generate Common Lisp code.

For example, after design and refinement, the specification of a transportation scheduling domain comprises about 10,000 lines of text of which about 3000 lines are the scheduling algorithm (the remainder consists of axioms and datatype operations that are not needed by the scheduler).

A key point here is that the high level of automation in Planware is achieved by applying domain-specific control (via a hand-built tactic):

1. to construct a problem specification and domain theory,
2. to apply a series of domain-independent design theories and code-generation rules. The result is a fast, correct, executable scheduler automatically constructed from the user's description of a scheduling problem.

In the next section, we provide a brief introduction to the specification and refinement formalisms in Specware. In Section 4, we describe Planware by stepping through its design process, illustrating each step via the construction of a transportation scheduler.

```

spec PARTIAL-ORDER is
  sort E
  op leq: E, E -> boolean
  axiom transitivity-axiom is
    leq(x, y) & leq(y, z) => leq(x, z)
  axiom reflexivity-axiom is
    fa(x: E) leq(x, x)
  axiom anti-symmetry-axiom is
    leq(x, y) & leq(y, x) => x = y
end-spec

```

Figure 2: Partial-Order SPEC in Specware

```

spec DRO-SPEC is
  sort D, R
  op O: D, R -> boolean
end-spec

```

Figure 3: DRO-SPEC in Specware

4. Specware

Specware supports the modular construction of formal specifications and the stepwise and componentwise refinement of such specifications into executable code. Specware may be viewed as a visual interface to an abstract data type providing a suite of composition and transformation operators for building specifications, refinements, code modules, etc. This view has been realized in the system by directly implementing the formal foundations of Specware; category theory, sheaf theory, algebraic specification and general logics. The language of category theory results in a highly parameterized, robust, and extensible architecture that can scale to system-level software construction. A more detailed description of Specware may be found in [14].

A *specification* (or simply a *spec* or *theory*) defines a language and constrains its possible meanings via (higher-order) axioms and inference rules. A basic specification consists of a list of sorts, operations and axioms. For instance, the theory of partial orders can be presented as an abstract sort with a binary operation that satisfies the following properties: reflexivity, transitivity, and anti-symmetry. A Specware spec for this theory is shown in Figure 2

Another example is a specification for a simple problem theory (DRO-SPEC) that consists of input domain, output range and a predicate that relates input to output (see Figure 3).

Specifications can be used to express many kinds of software-related artifacts, including application domain theories, formal software requirements, abstract data types, abstract

```

spec-morphism
INTEGER-AS-PARTIAL-ORDER:
PARTIAL-ORDER -> INTEGER is
{E-> integer, leq -> <=}

spec-morphism DRO-to-SORTING:
DRO-SPEC -> SORTING-SPEC is
{D -> set-of-integer,
 R -> sequence-of-integer,
 O -> sorting-pred}

```

Figure 4: Specification Morphisms

```

interpretation DRO-to-SORTING:
DRO-SPEC => SORTING-SPEC is
mediator SORTING-WITH-SORTING-PRED
dom-to-med
{D -> set-of-integer,
 R -> sequence-of-integer,
 O -> sorting-pred}
cod-to-med import-morphism

```

Figure 5: Interpretation in Specware

algorithms, formal interfaces for code modules, and so on..

A *specification morphism* (or simply a *spec-morphism* or *morphism*) consists of two specs and one mapping, that maps the source spec to target spec via sorts and operations maps such that the sorts map is compatible with the operations map, and moreover, axioms in the source spec are theorems in the target spec. For instance, a spec-morphism from the partial-order theory to integer can be represented by INTEGER-AS-PARTIAL-ORDER (in Figure 4).

Assuming that there is a spec SORTING-SPEC for the problem of sorting sequences of integers, a spec-morphism from DRO-SPEC to SORTING-SPEC can be expressed by DRO-to-SORTING (Figure 4), where we assume that the sorting specification SORTING-SPEC has a predicate `sorting-pred` to specify sorting requirements.

Specification morphisms underlie several aspects of software development, including the binding of parameters in parameterized specifications, specification refinement and implementation, datatype refinement, and algorithm design [10].

An *interpretation between theories* (or simply an *interpretation*) is a pair of spec-morphisms that essentially enables mapping an item to a term, which is what we need to express a refinement (or implementation) from one spec to another. Returning to our previous spec-morphism example with SORTING-SPEC, suppose that we do not have a predicate for `sorting-pred`, then it is impossible to map `O` to any predicate symbol in SORTING-SPEC.

However, we can map it to a term of SORTING-SPEC by forming a conjunction of all predicates that specify sorting requirements. This can be expressed via two spec-morphisms by DRO-to-SORTING (Figure 5). Here, we created a new spec SORTING-WITH-SORTING-PRED which imports SORTING-SPEC and adds another predicate sorting-pred that is defined from the predicates in SORTING-SPEC. In the scheduling domain, a scheduling spec has normally a list of constraints (some of them are provided by users, and thus the list is dynamically constructed). To start the refinement process on a scheduling problem spec (DOMAIN-SPECIFIC-SCHEDULING), we need to construct an interpretation from DRO-SPEC to DOMAIN-SPECIFIC-SCHEDULING, which will be given in detail in Section 3. It is indeed an interpretation since O in DRO-SPEC has to be mapped to the conjunction of all scheduling constraints present in DOMAIN-SPECIFIC-SCHEDULING. More precisely, an interpretation consists of two spec-morphisms: one from the source spec to the mediator spec, and another from the target spec to a mediator spec that is required to be a definitional extension of target spec. In the following, we will use the terms interpretation and refinement interchangeably.

In SPECWARE, the *colimit* operation is used to construct larger specs from smaller specs. The input to the colimit algorithm is a spec-diagram (a graph with nodes labeled by specs and arcs by spec-morphisms) called a base spec-diagram (also called a cover of that colimit/spec), and it computes a shared union of the specs in the spec-diagram. Colimits are used intensively in the construction and factorization of the scheduling domain knowledge base in Planware.

Given a spec, one refines it to more concrete specs via a sequence of refinements, so we need a sequential composition of interpretations to put these refinements together. Given a structured spec, for instance a spec formed via colimit, one only needs to give component interpretations of the cover, using parallel composition operator, a refinement for colimit object can be constructed automatically, provided the components interpretation are compatible with each other (this is where interpretation-morphism is used). Sequential and parallel compositions are used in the various Planware design tactics that will be described in Section 3.

Finally, after definitions have been created for all relevant sorts and operations in a spec, code can be generated, currently CommonLisp or C++ in SPECWARE. Code generation is carried out by means of logic morphisms, based on Meseguer's general logics [6] and their morphisms, with some modification. Our work in Planware of extending and applying Specware focuses basically on automating various combinations of sequential and parallel compositions, and representing knowledge.

5. Planware design process

Planware aims to provide a framework that is general enough to allow the synthesis of schedulers in a wide range of domains. The key to achieving this generality was our development of a specification for a generic scheduling problem that can be refined into a variety of concrete scheduling problems. Our confidence in this abstract scheduling specification

arises from experience with using the KIDS system to generate schedulers for such domains as transportation, manufacturing, power plant maintenance, satellite communications, pilot training, and others [11].

Briefly, here is how the Planware design process works. The user is asked to supply information about a particular scheduling problem. This information is used to refine the abstract scheduling specification to a specification of the user's problem. Planware then applies tactics that automatically perform problem reformulation and simplification, algorithm design, datatype refinement, expression optimization, and finally code generation. The following sections describe the steps in the Planware design process in more detail.

The most time consuming and novel aspect of this work is the automatic construction of a domain theory for the particular scheduling problem. In the KIDS system, this construction typically required weeks or months of time. In Planware this time is reduced to minutes, but for a sharply restricted domain.

5.1. Abstract scheduling problems

Abstractly, we consider a scheduling problem to be a set of reservations, where each reservation consists of a start-time, tasks to be accomplished and resources allocated. We do not specify the tasks and resources in detail, since these may vary from one scheduling domain to another. We specify an abstract scheduling problem as a function that takes a set of tasks and a set of resources as input, and returns a set of reservations (a schedule) that accomplishes all tasks and uses only the provided resources.

In the current system, the abstract scheduling spec is limited to problems of scheduling a single class of resource; e.g. scheduling cargo on aircraft, or scheduling the duty periods of personnel. Our next challenge is extending the abstract scheduling spec to allow multiple classes of resource, and the constraints on their interactions.

5.1.1. Abstract scheduling specification

The abstract scheduling problem is formulated as a structured spec. This structuring buys us: reusability, extensibility, implementability and evolutionary support of (re-)design. Basically, Planware's abstract scheduling specification has the following components:

Time – time is a total order,

Cap – capacity is a linearly ordered group,

Pre-Sched – a set of abstract reservations,

Res – an abstract resource spec,

Task – an abstract task spec,

Res-Sched – an abstract schedule for resource allocation,

Task-Sched – an abstract schedule that accomplishes tasks,

Sched-Base – a set of abstract reservations,

Scheduling – an abstract scheduler with two key constraints: all tasks are scheduled and only the provided resources are used.

This version is, however, a much simplified description of the actual spec in the system. The spec-diagram of Figure 6 shows their dependencies.

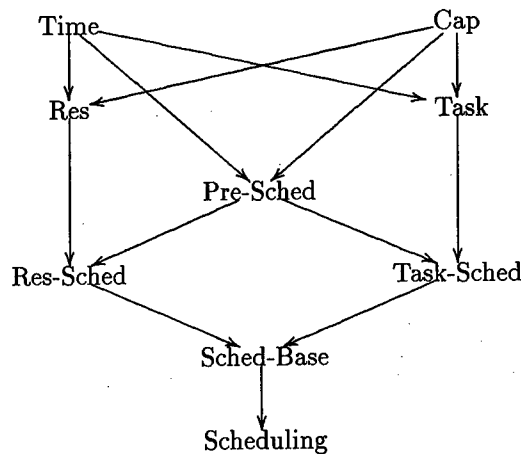


Figure 6: Scheduling System Architecture

Some of the specifications in this diagram are presented in Figure 7. Note how the attributes of Task and Resource are expressed as functions on those sorts (e.g. max-capacity). As new attributes are added under user guidance, we simply add new function symbols to the spec. Eventually, Planware refines Task and Resource to tuples and their attributes to projection functions.

5.1.2. Refining to a particular scheduling problem

Given the abstract scheduling specification, the very first step of refining to a given scheduling problem is to get information from the user about how to refine the resource and task components. For instance, a task in the user's problem may have a release-date and a due-date; if it is a transportation task then it may have an origin and destination. Currently Planware provides a taxonomy of task attribute specs for the user to select from. This taxonomy is straightforward to extend. Analogously, we have developed a taxonomy of resource theories from which the user selects. The next step is to obtain constraints on the scheduler from the user's choices.

A key goal of Planware is to free the user from the need to read or write formal specifications. To achieve this, we needed to find a way to lift information about tasks and resources into

```

spec RESOURCE is
  import TIME, CAPACITY
  sort Resource
  op max-capacity: Resource -> Quantity
end-spec

spec TASK is
  import TIME, CAPACITY
  sorts Task
  op task-demand: Task -> Quantity
end-spec

spec SCHEDULING is
  import SCHEDULING-BASE
  op Only-Avail-Res-Used: Resource-Set, Schedule -> boolean
  def of Only-Avail-Res-Used is
    axiom Only-Avail-Res-Used(resources, valid-sched)
      <=> in(a-reserv, valid-sched) => in(asset(a-reserv), resources)
  end-def
  op All-Tasks-Scheduled: Task-Set, Schedule -> boolean
  def of All-Tasks-Scheduled is
    axiom All-Tasks-Scheduled is
      All-Tasks-Scheduled(task-set, valid-sched)
        <=> in(task, task-set)
          => ex(a-reserv: Reservation)
            in(a-reserv, valid-sched)
              & in(task, tasks(a-reserv))
    end-def
  op Scheduler: Task-Set, Resource-Set -> Schedule
  axiom CONSTRAINING-SCHEDULER is
    Only-Avail-Res-Used(resources, Scheduler(task-set, resources))
    & All-Tasks-Scheduled(task-set, Scheduler(task-set, resources))
  end-spec

```

Figure 7: Scheduling Specifications in Specware

constraints on a scheduler. We observed that all of the constraints on tasks that we have dealt with can be characterized abstractly by means of a partial order. Intuitively, a feasible schedule of reservations must provide enough resource to meet the demand of the input tasks. This notion of meeting task demand particularizes to a partial order on each task attribute. For example, a due-date attribute on a task requires that the finish-time of its reservation be before the task's due-date (i.e. finish-time \leq due-date). For another example, the sum of the weights of the cargo items in a transportation reservation must not exceed the max-capacity of the transportation vehicle. Given partial order information about a task attribute, it is easy to create a constraint over an entire schedule; returning to the due-dates example: if valid-sched is the output of Scheduler(Tasks, Resources) then

```
fa(a-tsk: Task, a-reserv: Reservation, valid-sched: Schedule)
  (in(a-reserv, valid-sched)
   & in(a-tsk, tasks(a-reserv)))
  => finish-time(a-reserv) <= due-date(a-tsk)
```

Figure 8: Due-Date Constraint

In fact, we require that a task attribute be not only partially ordered, but that it have greatest lower bounds (i.e. be a meet semi-lattice). This requirement comes from the needs of algorithm design – the global search/constraint propagation algorithms perform fixpoint iteration in a semilattice.

By restricting to semi-lattice-structured task attributes, the task of constructing a formal specification of a scheduling problem, which is usually tedious and error-prone, is simplified to just asking the user to input/select whether each attribute is a lower/exact/upper bound. The corresponding constraints are constructed and asserted as output conditions of the desired scheduler. Additional work is required to add in the appropriate constructors and other operators for the refined datatypes of Task, Resource, Reservation, and Schedule. At this stage, Planware also constructs a slightly weakened version of the reservation and schedule specs, called Partial-reservation and Partial-schedule. These form the basis for the global search algorithm designed in a subsequent stage.

5.1.3. Example – transportation scheduling

In a simple transportation scheduling problem, the input tasks are movement requirements, which are descriptions of cargo that have to be moved. In this simple version a movement requirement includes information about when the cargo is available to be moved and by when it must arrive. So a schedule is a set of trips. Each trip has a start time and a manifest – the set of movement requirements that it has been assigned to execute.

The first phase of our development is to construct a transportation scheduling specification. Suppose we have enriched our resource taxonomy and task taxonomy to allow us to have basic transportation domain information like release-date and due-date, as well as the origin and destination of a trip, which are expressed by key words POD and POE, respectively. The Figures 9 and 10 show user selection interface.

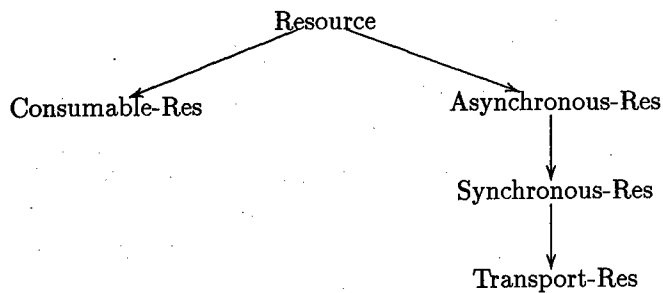


Figure 9: Resource Taxonomy

Choose Task Items
RELEASE-DATE
DUE-DATE
POE
POD
All of the Above
None of the Above
Do It
Abort

Figure 10: Task Taxonomy

Suppose we have selected transportation resource from the taxonomy and all task attributes shown in the above figure. Here, the colimit operator is used to put all task attributes together to form a domain-specific task spec. The parallel composition operator is used to put all domain-specific interpretations together to form a domain-specific scheduling spec.

The next step in the development process, if we choose to go forward, automatically constructs a transportation scheduling specification via the tactics described above.

5.2. Data-type reformulation

The construction process described above produces a scheduling specification for a particular problem. It is still formulated in terms of the schedule datatype which is a set of reservations. This formulation is general and supports the initial problem acquisition stage in Planware, but it is a relatively poor datatype for implementation purposes. In this stage, the Planware design process applies a datatype refinement that is stored with the resource theory that was chosen from the resource taxonomy. The effect is to refine $Schedule = set(Reservation)$ into a datatype that is better suited to the resource properties. The payoff is that we can then simplify away some of the problem constraints because they are effectively built into the schedule datatype. After Planware refines the schedule datatype, it invokes a context-dependent simplification tactic [9] to simplify the constraints.

5.2.1. Example – transportation scheduling

After finishing the refinement of abstract scheduling to transportation-scheduling, a series of refinements is carried out: adding complete constructors for transportation schedule data type; refining set of reservations to a map that maps a resource to its scheduled tasks (in sequence with increasingly start-time as ordering); etc.

The transportation scheduling problem uses a transportation resource which is a refinement of a synchronous resource (i.e. all reservations on a synchronous resource must be synchronized in the sense that two reservations must be either separated in time by at least some minimal amount or else simultaneous – starting and ending at the same time). Planware has refinements from `set(Reservation)` to `map(Resource, seq(Trip))` which effectively implements a schedule as an itinerary – for each resource we have the sequence of trips that it makes. The characteristic synchronization constraint is then simplified from a complex disjunction to a simple linear check over adjacent trips. For a typical input of 10,000 movement requirements, the original formulation will have several hundred millions ground disjuncts for the synchronization constraint, versus about 100,000 in the refined formulation.

5.3. Algorithm design

A design theory for an algorithmic concept can be represented as a formal specification [10]. Any particular instance of that design theory corresponds to an interpretation from it to a specification of the particular problem being solved. For instance, various interpretations from divide-and-conquer theory to a sorting specification correspond to various sorting algorithms, such as quicksort, mergesort or Batchers' sort. Design theories can be arranged in a refinement hierarchy with specification morphisms providing the refinement links; e.g. a hierarchy of algorithm theories is presented in [8]. The concepts and procedures described below are intended to automate the process of algorithm design by choosing a chain of algorithm design theories for a particular problem, and by constructing an interpretation from the chosen design theory to that problem. Thus, an algorithm for the specific problem is constructed.

The representation of our algorithm design framework can be illustrated by the diagram in Figure 11, let us call it an algorithm design cube, or simply the cube in this paper. The arrows in the cube represent the relationship between abstract theory and the concrete problem. Technically, the left square in the cube is a spec-diagram corresponding to the abstract algorithm design knowledge; the right square in the cube corresponds to the domain-specific problem and program scheme. The arrows in between are interpretations. Essentially, the design tactics described below are based on sequential and parallel refinement composition operators, as well as others. In the following the intended meaning of each arrow (and spec) and the way to construct them is described in detail.

At the very beginning, we have only the node DRO labeled with the abstract problem domain theory, as it can be seen in the above algorithm design cube. When a concrete (or domain-specific) problem specification is chosen (along with the main function to be developed), a problem domain specification can be extracted from it (and it is done via an extraction tactic

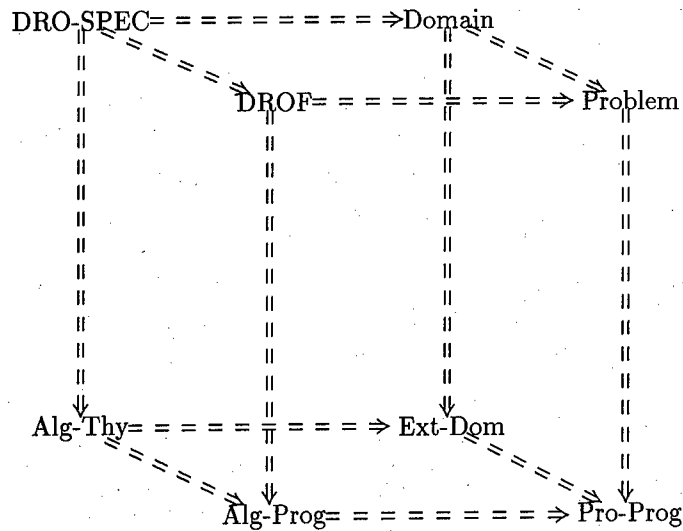


Figure 11: Algorithm Design Diagram

that gives an interpretation as result). So, the upper morphism on the right side of the cube is constructed.

The second arrow construction tactic, the domain-specific interpretation tactic, is a little more complex. The domain-specific interpretation tactic works as follows, first, use the main function signature to construct an interpretation from DRO to the problem domain specification. Second, using DRO and the main function signature, a DROF spec is chosen from the possible solutions specifications, e.g. all solutions spec, one solution spec and optimal solution spec, etc. Third, compute the colimit of the spec-diagram that include DRO, the domain-specific problem domain specification and DROF, which gives, among others, an interpretation from DROF to the colimit object. Finally, we check that the computed colimit is isomorphic to the domain-specific problem specification. In doing so, we have constructed and constructively proved that the base diagram of the computed colimit, namely, DRO, DROF and the problem domain specification is a cover of the domain-specific problem specification. Informally speaking, we can use DRO, DROF, and the problem domain specification to construct a program scheme, and that will be a program scheme for our specific problem too.

The third arrow construction tactic is called classification and it involves a process called ladder construction (see [8] for details). Here we only give a brief description of it in the context of algorithm design. Basically, this tactic consists of two steps: (1) selecting an appropriate design theory from a refinement hierarchy of design theories, and (2) constructing an interpretation. The first step is in general interactive, but can be automatic in certain domains (e.g. scheduling domains). The second step is accomplished via the incrementally constructing an interpretation, which is constructed by a constraint solving process that involves user choices, the propagation of consistency constraints, calculation of colimits, and constructive theorem proving. This is illustrated in the *ladder construction* diagram in Figure 12.

The result of the classification and ladder construction tactic is an interpretation from an

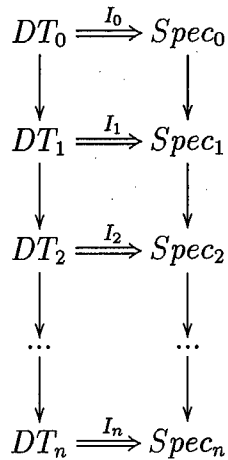


Figure 12: Ladder Construction

algorithm design theory to the problem domain.

The last tactic, the program scheme instantiation tactic, computes a colimit of the diagram that consists of algorithm design theory, its program scheme and the extended problem domain. The colimit object is domain-specific program scheme. Last but not the least, there must be a specification morphism from domain-specific problem spec to the constructed program. This is constructed and constructively proved to always exist by universal property of colimits.

With these four tactics, given a concrete problem, we can semi-automatically construct a program theory for that problem based on the selected and successfully interpreted algorithm design theory. Normally, further steps are needed to make it executable or more efficient.

The first design theory used in Planware is global-search theory and its extension with cutting constraints. Since this decision is fixed it is applied with no need for further interaction. Another algorithm design tactic used is constraint propagation. This amounts to generating basic constraint propagation procedures given a kind of scheduling problem domain, and synthesizing domain-specific constraint propagation procedures after the instantiation phase. Basically, that amounts to generating constraint propagation procedures for a set of upper bounds, exact bounds and lower bounds of domain-specific constraints. Technically, this is related to data type refinement to get the right constructors for each data type used in the constraints; and to the instantiation of the corresponding semi-lattices. After getting all the constraint propagation procedures, they are composed together and a flat semi-lattice is constructed that consists of a tuple of all component semi-lattices. Notice that this can only be done dynamically, since the constraint structure varies from one domain to another.

5.3.1. Example – transportation scheduling

The result of algorithm design is a domain-specific global search algorithm for the transportation scheduling problem (Figure 13), which is an instantiation of figure 11. That has cargo and pax capacity constraints, release and due date constraints, trip origin and destination constraints, as well as trip separation constraints.

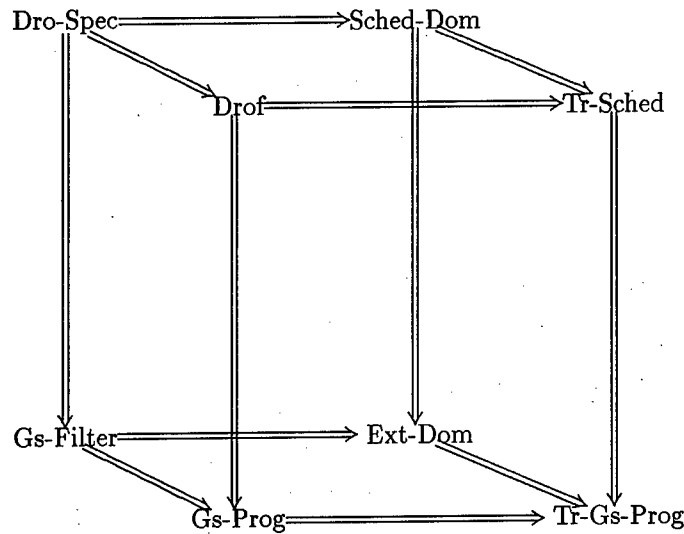


Figure 13: domain-specific algorithm

5.4. Expression optimization

This stage will apply various expression optimization refinements, such as context-dependent simplification, common-subexpression elimination, finite differencing, partial evaluation, and so on. These are not currently applied in Planware.

5.5. Automatic code generation

We have developed a code-generation tactic that automatically generates code for a structured spec, provided the structure is (recursively) of the following form:

1. directly implementable,
2. definitional extension or a translation of an implementable spec,
3. colimit, each component of which is implementable (recursively),
4. instantiation of implementable specs,
5. can be interpreted to an implementable spec.

If there are multiple choices, we use a heuristic to decide which way to go. In the Planware context, given the scheduling system architecture, we can generate code for it if each instantiated component is implementable, and we further specialize the code-generation tactic by a specific implementation order imposed by the dependencies of the scheduling systems architecture.

6. Synthesis of customized planners

Existing plan synthesis approaches in artificial intelligence fall into two categories – domain independent and domain dependent. The domain independent approaches are applicable across a variety of domains, but may not be very efficient in any one given domain. The domain dependent approaches need to be (re)designed for each domain separately, but can be very efficient in the domain for which they are designed. One enticing alternative to these approaches is to automatically synthesize domain independent planners given the knowledge about the domain and the theory of planning. As part of our research, we investigated the feasibility of using existing automated software synthesis tools to support such synthesis. Specifically, we developed an architecture called CLAY in which the Kestrel Interactive Development System (KIDS) is used to derive a domain-customized planner through a semi-automatic combination of a declarative theory of planning, and the declarative control knowledge specific to a given domain, to semi-automatically combine them to derive domain-customized planners.

The practicality of our approach is predicated on the availability of a software synthesis system capable of deriving code from formal specifications. KIDS is a powerful semi-automated system for development of correct and efficient programs from formal specifications. Given a domain theory and the input/output specification of a task, KIDS system helps in synthesizing a program capable of solving the task. The input to KIDS is a task theory comprised of the task specification and a declarative description of useful concepts and rules to reason in the task space. In this research, we give planning as a task to KIDS and expect it to synthesize and return a planner as the solution. The planner can then take planning *problems* as input and return *results* (plans).

In order to support planner synthesis, we have to develop and input a theory of planning to KIDS. Traditional plan synthesis techniques can be described in terms of a common plan representation, with different planners corresponding to different ways of refining the partial plans such as progression, regression and plan-space refinements. Consequently, our planning theory will consist of a specification of the planning task (in terms of input and output data types) and one or more refinement theories. Since we are also interested in domain-customized planners, we have to provide the necessary domain knowledge to KIDS.

Given these inputs, KIDS semi-automatically synthesizes a program (in this case, a domain dependent refinement planner) using generic algorithm design tactics (such as branch and bound, global search). The resulting planner, like conventional planners, can handle any planning problem from the domain.

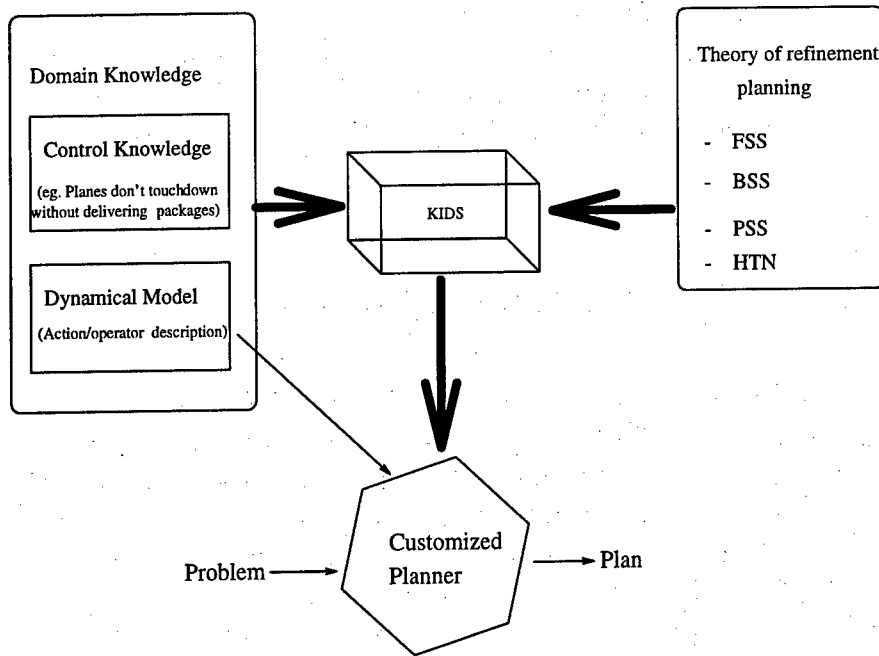


Figure 14: Architectural overview of planner synthesis with KIDS in the CLAY approach. The theories of refinement planning and domain knowledge are declaratively specified to KIDS which in turn combines them to produce a customized planner for the domain. The resulting planner, like conventional planners, can handle any planning problem from the domain.

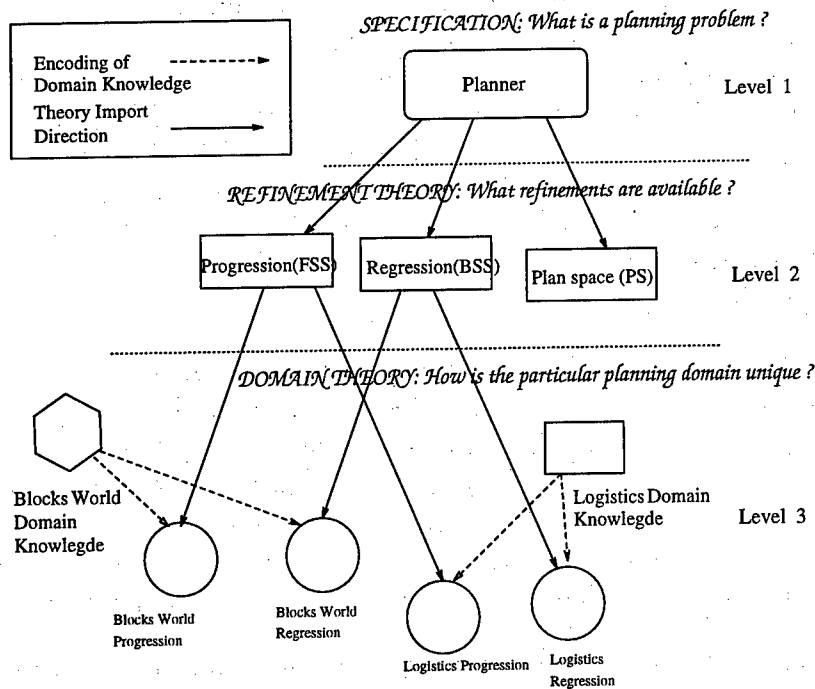


Figure 15: The CLAY architecture for writing planning theory. Each level answers a question relevant to that level of planning detail. CLAY uses KIDS' feature of theory import to modularize the domain-specific planning theory

To understand the efficacy of plan synthesis in CLAY, we concentrated on the synthesis of planners using state-space refinement theories.¹ Empirical evaluation shows that these synthesized planners can be very efficient. For example, in the blocks world domain where the goal was stack inversion, a KIDS synthesized planner solved a 14 blocks problem in under a minute. In the logistics domain, a problem with 12 packages, 4 planes and 8 places was solved in under a minute. Similarly, in the Tyre domain, the "fixit" problem was solved in under a minute. To put these performance results in perspective, we compared KIDS' synthesized planners to a set of classical planners implemented as the instantiations the UCP planning system. In our experiments, the best of the KIDS' synthesized planners outperformed the best of the UCP instantiations when given the same domain-specific information. We hypothesize that this is because KIDS can profitably fold-in the domain-specific control knowledge (i.e., the domain theory) into the planning code.

Our approach strikes a promising middle-ground between domain independent and domain dependent planners. The theories of planning are encoded independent of domains, and the domain control knowledge can be encoded independent of the specific planning theory being used. The customization step compiles the domain control knowledge into the planning algorithm and ensures that the resulting planners are able to exploit the structure of the domain.

Results of this work have been published in the Journal of Artificial Intelligence Research, as well as in the Proceedings of 1997 Automated Software Engineering conference.

7. Concluding remarks and future work

We have presented our Planware system for generating domain-specific high-performance scheduling software in a highly automated way with minimal user input. Planware is an extension of the Specware formal development environment. Scheduling domain knowledge has been represented abstractly and formally to enable user problems to be solved with minimal interaction. In particular, the resource and task taxonomies which specify general/domain-specific scheduling knowledge have been developed as well as their architectural relationship with the scheduling system architecture. For synthesizing domain-specific schedulers, a set of design tactics for instantiation to the concerned problem, data-type refinement, algorithm design with constraint propagation, and automatic code-generation have been developed and successfully applied. We have experimented with the transportation scheduling domain and developed a variety of schedulers.

We believe that Planware is a new paradigm for domain-specific software generators. Planware differs from other domain-specific software generators in that it is built on a foundation of domain-independent general-purpose software specification and synthesis capabilities (Specware/ Designware). In particular, Planware relies on

1. the Specware capabilities for composing specifications, refining them and translating to code;

¹In future, we plan to extend our approach to plan-space and task-reduction refinements.

2. the Designware libraries of domain-independent design knowledge about algorithms, datatype refinements, and expression optimization techniques (and their application tactics) to construct refinements.

The domain-specificity of Planware comes in the form of

specifications of domain knowledge – the abstract scheduling specification, the taxonomies of task and resource theories, etc.

scheduling spec construction tactics – tactics for lifting properties of tasks to constraints on the scheduler, tactics for lifting resource constraints to scheduling constraints, tactics for constructing the constructors and other datatype operations needed by the refined Task, Resource, Reservation, and Schedule specs,

embeded algorithm design tactic – tactics for generating a global search theory for the scheduling problem at hand, etc.

The background of domain-independent design knowledge allows a user to derive software even when the requirements fall outside the domain-specific scope of the system. The user then gets less automation, and must supply more guidance in the construction process.

There are many things to be done before Planware can be deployed. One crucial extension is allowing the user more flexibility in supplying task information. We are developing a spreadsheet-like interface that is derived from the user's choice of resource theory and presents the user with plausible options for lower/exact/upper bounds on all crucial attributes of a reservation for that kind of resource. We are working to let the user choose and modify arbitrary entries. As before the user only interacts with the system in domain-specific terms. Another vital extension is to generalize the abstract scheduling spec to multiple resource classes. Another extension that is underway is to extend Planware to allow the synthesis of scheduling systems, including visual displays, editors, GUI, database mediators, and so on.

References

- [1] APPLGATE, D., AND COOK, W. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing* 3, 2 (Spring 1991 1991), 149–156.
- [2] BLAINE, L., AND GOLDBERG, A. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.
- [3] FOX, M. S., SADEH, N., AND BAYKAN, C. Constrained heuristic search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (Detroit, MI, August 20–25, 1989), pp. 309–315.

- [4] FOX, M. S., AND SMITH, S. F. ISIS – a knowledge-based system for factory scheduling. *Expert Systems 1*, 1 (July 1984), 25–49.
- [5] LUENBERGER, D. G. *Linear and Nonlinear Programming*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1989.
- [6] MESEGUER, J. General logics. In *Logic Colloquium 87*, H. E. et al., Ed. North Holland, Amsterdam, 1989, pp. 275–329.
- [7] SCHANK, J. *A Review of Strategic Mobility Models and Analysis*. Rand Corporation, Santa Monica, CA, 1991.
- [8] SMITH, D. R. Toward a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96* (1996), vol. LNCS 1101, Springer-Verlag, pp. 62–84.
- [9] SMITH, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.
- [10] SMITH, D. R., AND LOWRY, M. R. Algorithm theories and design tactics. In *Proceedings of the International Conference on Mathematics of Program Construction, LNCS 375*, L. van de Snepscheut; Ed. Springer-Verlag, Berlin, 1989, pp. 379–398. (reprinted in *Science of Computer Programming*, 14(2-3), October 1990, pp. 305–321).
- [11] SMITH, D. R., PARRA, E. A., AND WESTFOLD, S. J. Synthesis of planning and scheduling software. In *Advanced Planning Technology* (1996), A. Tate, Ed., AAAI Press, Menlo Park, pp. 226–234.
- [12] SMITH, S. F. The OPIS framework for modeling manufacturing systems. Tech. Rep. CMU-RI-TR-89-30, The Robotics Institute, Carnegie-Mellon University, December 1989.
- [13] SMITH, S. F., FOX, M. S., AND OW, P. S. Constructing and maintaining detailed production plans: Investigations into the development of knowledge-based factory scheduling systems. *AI Magazine 7*, 4 (Fall 1986), 45–61.
- [14] SRINIVAS, Y. V., AND JÜLLIG, R. Specware: Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin, 1995, pp. 399–422.

***MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)***

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.