AFRL-IF-WP-TR-2004-1528

AIRES: AUTOMATIC INTEGRATION OF REUSABLE EMBEDDED SOFTWARE, METHODOLOGIES, TOOLKIT, AND EXPERIMENTS

Kang G. Shin, Shige Wang, Zonghua Gu, Sharath Kodase, and Jeong-Chang Kim

University of Michigan
Real-Time Computing Laboratory
Ann Arbor, MI 48109

FEBRUARY 2004

Final Report for 01 June 2000 – 28 December 2003

STINFO FINAL REPORT

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

NOTICE

Using 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 neither responsibility nor any obligation whatsoever. The fact that the government formulated or in any way supplied the said drawings, specifications, or other data does, 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 has been reviewed by the Office of Public Affairs (ASC/PA) and 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.

/s/

CHARLES P. SATTERTHWAITE, Project Engineer
Embedded Information System Engineering Branch
AFRL/IFTA

/s/

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

/s/

EUGENE C. BLACKBURN, Branch Chief
Information Technology Division
AFRL/IFTA

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, BLDG 620, 2241 AVIONICS CIRCLE, WRIGHT-PATTERSON AFB, OH 45433-7334 TO HELP US MAINTAIN A CURRENT MAILING LIST.

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

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
February 2004	Final	06/01/2000 – 12/28/2003

4. TITLE AND SUBTITLE
AIRES: AUTOMATIC INTEGRATION OF REUSABLE EMBEDDED SOFTWARE, METHODOLOGIES, TOOLKIT, AND EXPERIMENTS

5a. CONTRACT NUMBER
F33615-00-C-1706

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
69199F

6. AUTHOR(S)
Kang G. Shin, Shige Wang, Zonghua Gu, Sharath Kodase, and Jeong-Chang Kim

5d. PROJECT NUMBER
ARPI

5e. TASK NUMBER
FT

5f. WORK UNIT NUMBER
0G

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Michigan
Real-Time Computing Laboratory
Ann Arbor, MI 48109

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Information Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7334

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/IFTA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-IF-WP-TR-2004-1528

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Report contains color.

14. ABSTRACT

Program objective is to develop methodologies using a combination of object-oriented models, formal methods and real-time computing principals for model-based integration of embedded real-time software under assumptions that (1) there exists reusable software components; (2) timing characteristics of components under different resource constraints are known or can be measured experimentally; and (3) all application functionality and timing can be specified at design phase.

15. SUBJECT TERMS
modeling and simulation, object oriented, formal methods, reusable software, embedded real-time software

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Charles P. Satterthwaite
Unclassified	Unclassified	Unclassified	SAR	112	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-6548 x3584

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Abstract

The Automatic Integration of Real-Time Embedded Software (AIRES) project aims to develop modeling methodologies, analysis algorithms, and software development tools with engineering processes built in, to support the integration of embedded software components subject to non-functional (e.g., timing and scheduling) constraints. In this project, we have proposed, implemented, and evaluated a model-based approach for embedded software construction and analysis. Our approach is based on meta-modeling and model-transformation techniques. The meta-modeling technique allows us to define the model constructs required to model and analyze the non-functional properties of embedded software. Using the meta-model, one can specify not only software components and applications built with the components, but also their execution environment or platform, non-functional constraints, and performance-related parameters. Model transformation supports automatic evolution of design models from one phase to another within the development process. The AIRES project focused on the transformation of a structural model to a runtime model. To ensure that the models are transformed while meeting the timing and scheduling constraints, we have developed a set of algorithms, including component allocation, task formation, assignment of timing attributes and scheduling policies, and timing and schedulability analysis. All of the thus-obtained components, called the *AIRES toolkit*, have been implemented in the Generic Modeling Environment (GME), a Windows-based graphic modeling environment designed by researchers at Vanderbilt University. The AIRES tool kit has been evaluated in the form of an end-to-end integration tool-chain for automotive and avionics applications. Together with other research groups that participated in the evaluation, the AIRES tool is shown to achieve a 2–10 $\times$ speedup in locating design errors and recommending solutions to fix them. The speedup also comes from the automatic process and algorithms built into the tool which find optimal/suboptimal solutions in a large design space.

Contents

Chapter 1

Overview

The AIRES project research has been conducted under the auspice of the DARPA/IXO Model-Based Integration of Embedded Software (MoBIES) Program. The goal of the MoBIES program is to develop mathematical models and interface standards for the integration of physical descriptions of application domains, high-level specifications of program functionality, and process models for software tools. Using this technology, one can construct a domain- and application-specific embedded software design frameworks with different tools. The tools can be developed using different modeling languages that are specialized for different design concerns and different modeling aspects of the target embedded software. Therefore, various cross-cutting issues in embedded software design and analysis, including real-time scheduling, size/weight/power limitations, fault-tolerance, safety, and security, can be addressed by using the various tools in the integrated tool chain. Such customizable tool integration frameworks will dramatically accelerate the embedded software development process, and yield correct and high-quality code.

As a part of the MoBIES program and as a significant contribution to end-to-end tool chain integration, we have developed techniques and tool kits specifically to address the issues of software modeling, design, and analysis related to timing performance and schedulability.

1.1 Motivations and Project Scope

Today's embedded software design tends to synthesize existing software components and models for fast and low-cost software development. In such a describe-and-synthesize development methodology, the design process of embedded software selects the software models/components that can meet both functional and non-functional constraints. Since embedded software typically runs on a resource-limited platform, and is responsible for correct control of the external physical world, meeting non-functional and functional constraints, such as timing, schedulability, power, and size, becomes a critical issue. In view of the large and complex non-functional constraints in embedded systems, we focus on the design and analysis issues associated only with timing and scheduability.

Embedded applications can be classified as *computation-intensive* or *communication-intensive*. A computation-intensive embedded application determines the status of the external physical world and makes control decisions based on complex computation and data processing. Examples of such applications include the mission control of weapon systems, automotive in-vehicle control, and machine-tool control applications. A communication-intensive embedded application determines the system status and generates control commands mainly based on collaboration among a large number of devices. Examples of this type of applications include sensor networks and home automation systems. The software for computation-intensive embedded applications typically runs on a small number ($(\leq 10^2)$ of computation devices. However, a large number of software components/algorithms run on each of these

devices. In contrast, the software for communication-intensive applications usually runs on a large number ($10^2 \sim 10^4$) of devices, each executing simple operations. The AIRES project aims to address software-development issues for computation-intensive embedded applications.

To support fast, low-cost, and high quality software development for computation-intensive embedded applications, we should address the following modeling and analysis problems.

Integrated performance modeling framework. Support for performance modeling with the functional system modeling is essential for early design error detection and decision on design choices. To this end, a performance modeling framework needs to be defined and associated with the functional design framework, including modeling methods, performance parameters and metrics, and interfaces with other models and analysis algorithms. The thus-defined performance modeling framework should be sufficient to cover various modeling and analysis requirements for different models and different development phases, and should be flexible for use with different functional modeling and analysis techniques. It should be customizable for different modeling and analysis requirements since only limited knowledge can be provided for a given design aspect, and only limited information can be manipulated by the designer.

Automatic mapping between models with consideration of performance constraints. The software design of an embedded system is typically a multi-stage process. At each stage, a system model is generated from the model used in its preceding stage by refining with more design details that became available. Current practice for such model transformation is a manual trial-and-error process. A challenging problem is how to use the performance constraints to guide the design so that only those transformations that will lead to meeting the constraitns are considered. Moreover, automation of model transformation is always highly desirable, as it can not only accelerate the development, hence reducing the development cost, but also eliminate errors introduced by manual model transformation.

Performance analysis of designed software in distributed environments. Performance analysis is critical to ensure the runtime correctness of the designed software. The performance analysis determines: (1) if the performance constraints can be met, and (2) how to configure the system so as to meet all constraints. In a distributed environment involving multiple devices, it is very difficult to verify that all performance constraints are met since it involves modeling all possible dynamic scenarios at runtime. It is even more difficult to determine the configuration that meets all constraints.

Performance-aware design process. The software design process for embedded software will change with system performance considerations. Instead of only checking if functional requirements are met at each design stage, we must now check both functional and performance constraints. This will result in a performance-aware design process which finds a software configuration that fullfils the required functions with the desired performance at each stage. We build such a design process into a development tool chain to support the evolution of the software design model. A difficult issue to be addressed in this process is information dependencies between different development phases. Such information dependencies are critical, especially for performance analysis. An example information dependency in embedded software design is allocation of software components to physical devices to meet performance constraints that depends on the components' resource consumptions, which in turn depends on the device that each component is allocated to.

Integration of software development tool-chain introduces more issues than those listed above. As embedded software becomes more complex, it is difficult to design, implement, and verify the software

without development tool support. Generally, the software tools used at different development phases include tools for modeling, analysis, and implementation. Different tools could be used at a certain design phase to process different aspects of the software. Different design phases may also require different tools. As these tools are built with different system modeling assumptions, it is difficult, if not impossible, to make these tools cooperative to solve the software development issues as a whole.

In current practice, the software for an embedded control system is first designed to meet its functional requirements. Such a functional design includes components/operations for data acquisition, data processing, control algorithms, and control commands generation. It also includes device drivers for sensors and actuators. These functions are then partitioned into different subsystems for implementation. The performance of the designed software is analyzed after the design details are filled in. Meanwhile, the platform for running the software is determined based on the system capacity planning. The software components and subsystems are allocated on the platform while considering the performance constraints. This is currently done by trial-and-error, and is one of the main bottlenecks in the software development process. The runtime model is then generated with all timing attributes assigned.

Our objectives are to identify the modeling requirements of performance-aware embedded software design, develop methodologies and algorithms to support performance-aware design and integrated system analysis, and demonstrate that the developed methods and algorithms can be integrated with other models in a modeling framework that meets the modeling requirements in a tool-chain integration.

1.2 A Project Overview

The techniques we developed are to address the challenging problems related to performance modeling and analysis. To address the performance modeling framework issues, we adopted a meta-modeling framework. A meta-model defines the elements and their inter-relationships for building application models. Our performance information is annotated to the functional meta-model components. As different meta-model components may be used for different aspects of the system design and modeling, these components may have different performance information associated with them. Annotation of performance to functional meta-model components allows the designer to specify and manipulate the performance related only to those aspects of interest. Meanwhile, the performance parameters and their corresponding values can be extracted by accessing functional components in an application model, and hence, the performance manipulation can be easily integrated with the functional manipulation, and accessed when a functional component is accessed. It is unnecessary to develop a dedicated algorithm for performance information extraction from the model, thus significantly simplifying both algorithm implementation and tool integration. Given the meta-model-based performance modeling framework, we have developed a set of algorithms for automatic model transformation algorithms. These algorithms take a high-level design model, filling more implementation details in such a way that the performance constraints can be satisfied. As the design model is normally expressed as a graph or a set of graphs, our algorithms employ the graph transformation techniques to refine the design model. The overall performance constraints are then verified for the detailed models to detect any potential violation of the constraints. The analysis algorithm verifies the satsifaction of constraints using the busy period concept in the well-known real-time scheduling theory, and provides information on resource consumption as well as individual delays of all components and the system. To break the information dependencies among different stages of the software design, some assumptions are made initially for the model transformation. An iterative process is developed to correct/refine the results obtained under the initial assumptions.

To demonstrate the correctness and effectiveness of the developed components, we have implemented all of the above-mentioned techniques and algorithms as the AIRES toolkit. This toolkit is integrated with the Generic Modeling Environment (GME) tool from Vanderbilt University. GME is a MS Windows-based graphic tool, which provides powerful mechanisms to define meta-models and the

programs that manipulate models based on the meta-model. All of our algorithms are implemented as plug-ins in GME, which is called the *interpreter*, and implemented as MS Windows Dynamic Link Library files (DLL). The software construction process is also built into the tool as the interpreter invocation sequence. The precedence between the interpreters reflects the information dependencies between design models at different phases, and results can be refined by re-applying the algorithms to a thus-obtained model.

To meet different design and analysis requirements in different domains, we have tailored the tool for use in both avionics mission computing and automotive vehicle control applications, which were selected as the DARPA MoBIES Open Experimental Platforms (OEPs), to demonstrate the flexibility and effectiveness of the AIRES tool. Our results have shown that for both OEPs, the AIRES tool can be easily integrated with other functional design and modeling tools, detect performance errors, and generate (performance-wise) better system configurations. Our evaluation has also shown that the software development process is accelerated by integrating the AIRES tool in the development tool-chain for large and complex embedded systems.

1.3 Organization

The rest of the report is organized as follows. Section 2 discusses the modeling methods and models that we use in the AIRES toolkit. This includes performance models with meta-modeling and meta-models for both automotive and avionics applications. Section 3 presents the algorithms we have developed to analyze the software application models. These algorithms were designed based on the information defined in the meta-model, and were used for both design automation and system analyses. Particularly, we have developed algorithms for functional verification such as signal composbility and event/invocation dependencies, algorithms for runtime model generation including component allocations, timing assigments, and task formation, and algorithms for schedulability and timing analysis. Section 4 describes the end-to-end measurement method developed to obtain the underlying system overheads. The method uses synthetic workloads and microbenchmarks with a sampling measurement tool, and provides realistic analysis results and critical information for platform evaluation. Section 5 details the implementation of AIRES toolkit, including the models, algorithms, and methods developed in this project. We have taken two different approaches to implementing these techniques in order to show that the AIRES tool can be used as a standalone tool as well as a participant in a tool-chain. Section 6 presents the evaluation results of both individual algorithms and the integrated experiments. The evaluation of individual algorithms has demonstrated the scalability and accuracy of the analysis results of all developed algorithms, while the integrated experiments have demonstrated AIRES's usefulness and effectiveness for the whole software development process. The report concludes with Section 7.

Chapter 2

Modeling Language

To support timing and schedulability analysis, one must specify essential information with the functional model. In this project, we annotated the information essential for analysis. The annotated information includes both *timing requirements* and *component charateristics*. Further, to simplify the analysis, the functional models are also categorized to define different aspects of the designed software along the development process. The following sections describe the organization of the model elements, annotated timing related information, and thus-tailored models for the domains of two OEPs.

2.1 Modeling elements and organization

We modeled different aspects of the functional system and performance. We classified models into 4 catergories according to their usage in the different design stages: (1) the *component model* is constructed first before the software structure is defined, and is used as basic building blocks in the software construction; (2) the *platform* model specifies the execution environment of the software; (3) the *software model* defines the software architecture; and (4) the *runtime model* specifies the runtime structure of the system.

Software component model. To support the analysis, a software functional model must be constructed first. Such a model is usually in some form of functional abstraction, and can be customized for a family of applications in the same domain. Examples of such a component include devices components, data processing components, and control algorithms.

The structure of each functional component is modeled as a port-based object, as shown in Figure 2.1. In this model, each port-based object consists of a set of ports for inputs and outputs, a set of actions for computation, and a behavioral specification to define component behaviors under different system modes. The ports of a component provide a unified mechanism for a component to interact with other components in an integration. The ports can be mapped to various process communication mechanisms supported on a platform, such as global variables, shared memory, and local/remote message passing. Each port can be associated to one or more types of events, therefore transforming the component to a typed system for formal verification. The action set defines computations that a component can perform. For example, a target positioning component can determine the target position in either 2-dimentions or 3-dimentions, depending on the operation mode. Which actions the component will perform under which mode is defined in the behavioral specification. Such behavioral specfications can be either static built-in the component, or customized when the component is instantiated for in an integration. Note that a component without event types associated to its ports and behavioral specifications defined is only an abstract one. The formal verification is only applicable after an integration is constructed when all components have been instantiated.

Formally, a software component can be defined as follow.

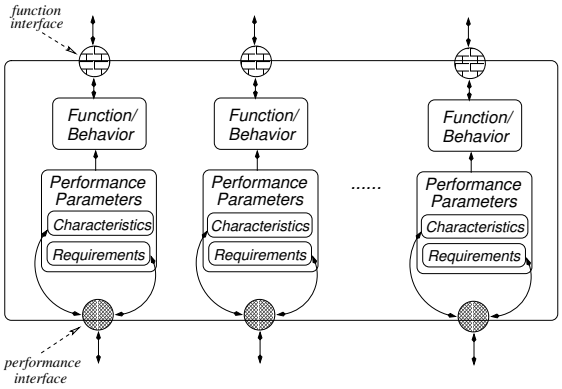


Figure 2.1: Component structure.

Definition 1 *A* software component *is a pair* $c =< B_c, P_c >$*, where* B_c *is a set of actions to transform inputs to outputs controlled by a behavior specification, and* P_c *is a set of ports for inputs* (I_c) *and outputs* (O_c) *such that* $P_c = I_c \bigcup O_c$ *and* $I_c \bigcap O_c = \emptyset$.

The component model in this work is more *process-centric*, meaning the component performs a single type of function. To this end, a component can be viewed as a function call with input ports and output ports are input and return parameters. An example of the *process-centric* component can be a locating target component, which determines the relative position of a target according to the sensed data. On the other hand, many applications adopt *data-centric* components. In a *data-centric* component, ports are set of methods or function calls. A *data-centric* component usually performs different functions according to which port is activated. For example, an engine component may perform ignition operation and/or idle operation according to which function is called. We argue that the *process-centric* components are finer grain components than the *data-centric* components, and a data-centric components can be split into several process-centric component. Further, when data-centric components are used to construct software, simultaneous invocations of different functions of one component are possible. Since these functions may operate on the same set of data built into the component mutual exclusion is required. This is implicitly modeled, which can make the analysis difficult. To eliminate these difficulties, we use a port-depedency diagram to convert a model with data-centric components to one with process-centric components. Details are given in later discussion on software model.

Performance of a component is modeled as a set of annotated attributes of actions. Since components are building blocks in our integration model, the system-level performance must be derived using the performance constitute components (along with their interactions and execution environments). Although the overall component performance varies according to the event it handles and behaviors it performs, the performance of each action, such as worst-case execution time, is almost a constant for a given platform. Further, some components may be subject to certain performance requirements. For example, a data collection component of a continous motion must be invoked at a certain frequency. Such requirements should also be annotated to the actions. Figure 2.2 shows the component structure with performance annotations.

Platform model. A platform defines the execution environment of the designed software, therefore, the model of the platform is essential for software analysis. A platform consists of components of hardware such as processor, memory and communication links, and supportive software such as operating systems and middleware. The system can then be viewed as a hierarchical realization model as described in [2]. A 3-layer realization hierarchy is showed in Figure 2.1.

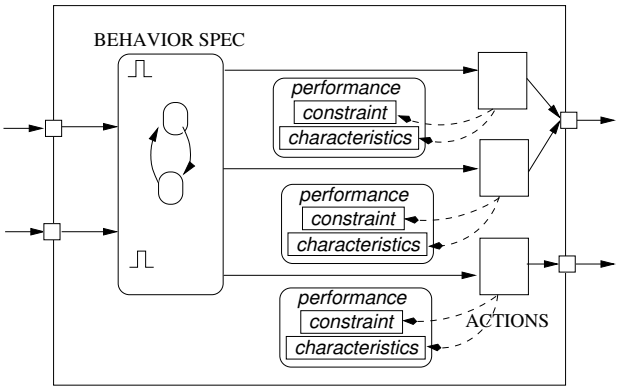


Figure 2.2: Component structure with performance annotations.

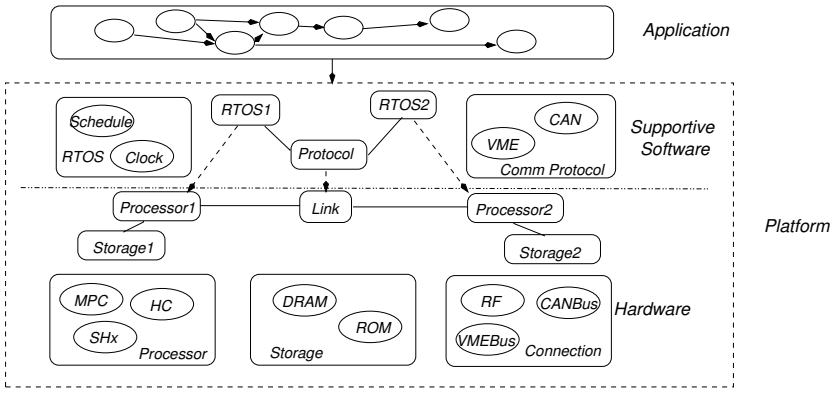


Figure 2.3: Hierarchical realization model.

In this realization hierarchical model, the modeled platform components include hardware components and supportive software components. The hardware components include various types of processors, memory modules, and connection components. The performance characteristics of these components include information such as the clock speed of a processor, memory access time, propogation delay and transmition speed of a connection. To simplify the work yet make it representative, we considered only operating systems for the supportive software, and assumed the application will be running directly on top of the real-time operating systems (RTOS). The RTOS is also modeled as a collection of services, such as scheduling service, timing and clock management service, interrupt services, etc. Performance of each service is also modeled as RTOS component performance in the platform model. The performance includes delays and variances introduced by these services (since they share the same hardware resources with application components). Note that the performance of the supportive software may vary according to the workload and configuration of the application software. We developed an end-to-end measurement-based technique to quanttitatively profile these effects in this work, which will be discussed in more detail in later section of this report.

Software model. The designed embedded control software is modeled as a set of intercommunicating components. Each component has a structure as defined in the component model, and performs a portion of the functions defined in the overall system design. The software with constitute components and their communications can then be represented in a form of directed graph, called a *structural model*. Each node in the structural model represents a software component, and each edge represents

a communication between a pair of components. Note the communication can be a data-flow message or a control flow event. Figure 2.4 shows an example of software model.

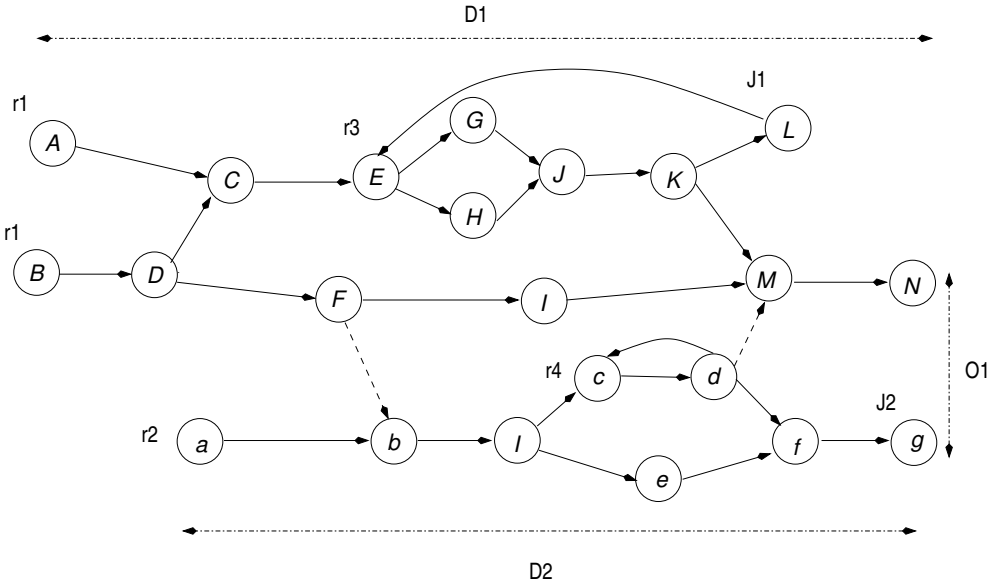


Figure 2.4: An example software model.

The behaviors of the designed software are represented as component chains, called *transactions*, in a structural model. To facilitate the analysis, each transaction is supposed to be a directed acyclic graph. For a transaction that contains a cycle, it can be converted into acyclic one as follows:

1. The contained cycle runs at a faster rate than the transaction. This is typical in a multi-rate control system, where an inner control loop runs faster than the outer control loop. In this case, the cycle representing the inner control loop can be modeled as a separate transaction, and is subsituted with a single node in the original transaction. The graph after the conversion is semantically equivalent to the original one because (i) communciations in such a case must be data communication (a control loop will force the cycle to run at the same rate as the transaction), and (ii) not all data generated by the cycle can be consumed by the transaction.

2. The contained cycle runs at the same rate as the transaction. This is typical for close loop feedback control. In this case, there must be at least one communication in the cycle that is data communication. Otherwise, the cycle forms a livelock, and exhuasts the computation resource for other components in the transaction to run. We can then break the cycle from the data communication edge, making the the generated data in the $i-th$ invocation to be used in the $(i+1)th$ invocation. Then the cycle is broken into a chain in the transaction, and runs at the same rate.

In the software model, components can be assigned to an execution location. An execution location is a processor that the component will reside on. We allow software components bound to certain processor in the software model in order to support modeling device- and location-dependent software, such as I/O drivers and data preprocessing software. The constraints of software component execution locations are called *location constraints*. It is the software designer's responsibility to model the location constraints in a software model. In this work, we assume all components are statically allocated to some processor,

and will not migrate during execution. There can be multiple copies of a component for fault-tolerant and reliability reason. However, we assume that at most one of these copies is allowed to be active at a given time, and all copies maintain a consistent system state.

For a model constructed with data-centric components, we convert the model to one with process-centric components. The model with data-centric components is constructed by converting the model to a port dependency graph (PDG) as follows:

1. For each method/function of a data-centric component in the model that is invoked by other component(s), create a node in the PDG.
2. For each precondition a PDG node needs to satisfy for its execution, including both events and data, create an input port for it.
3. For each results a PDG node generates, including both events and data, create an output port for it.
4. For each invocation link in the original model, create a link between corresponding ports of nodes in PDG.
5. If there exist multiple event or data links between a pair of nodes in PDG, merge them into one.

The thus obtained PDG is a model with process-centric components.

The performance specifications for the software model include both performance constraints and performance characteristics of individual software components and communication links. At the system level, the rate of each transaction and its completion deadlines must be specified. The constraints among the transactions, such as input separation [1] and output correlations [2], should also be modeled. At the component level, the resource demand, usually represented as the worst-case execution time, of a component must be provided. In our model, the component resource demand is modeled as a cost of a node. Additionally, the resource demand of a communication link between components, represented in communication delay, should also be provided. The resource demand of a communication is modeled as the cost of the link in the model. Depending on the type of the communication (data or event) and the size of the message passed over the link, the link cost are different. Usually, we assign the cost for an event link to zero as such delay is negligible (function call takes trivial time), and assign the cost for a data link to its maximum message size.

Runtime model. A runtime model specifies the runtime structure of the designed software. As more and more embedded software tend to use the commercial embedded and real-time operating systems for the runtime schedule and resource management, and the basic schedulable unit in an operating system is a process/thread (called a task), the design software with all components and inter-component communications must be mapped to tasks for scheduling and executions on a target platform.

The runtime model is usually represented as a task graph, which is a directed acyclic graph with each node in the graph as a task, and the link between nodes for task precedent constraints (data and/or event). To schedule a task, the system needs to know the task invocation rate, the resource demands of a task in the form of the worst-case execution time, and the task's priority to determine the resource allocation when there are multiple tasks competing for the same resource. For a runtime model, each task must also be assigned a processor for execution.

[1] An input separation is defined as the time difference between to input data. Satisfaction of input separation constraints will ensure the transaction using correct version of input data to generate the outputs.

[2] An output correlation is the time difference between two outputs. Satisfaction of the output correlation constraints ensures the stable system and correct control.

The performance specifications for a runtime model include both specification for tasks and for communications. The task performance specfication includes task invocation rate (period) and its worst-case execution time. The task invocation rate are derived from the transaction rate in the software structural model. The worst-case execution time of a task can be computed using the resource demands of the components inside the task.

2.2 Meta-model construction

The implementation of the AIRES meta-model follows the discussions given in the previous section. Particularly, the meta-model consists of 4 modeling categories, namely the component folder for reusable components, software folder for software structure of an application, hardware folder for platform configuration, and task folder for runtime system modeling. In the context of MoBIES program, we applied and evaluated the modeling techniques to MoBIES selected OEPs of Automotive and Avionics applications. Due to the differences of modeling requirements, development tool chain, and development processes for Automotive and Avionics applications, we tailored the above discussed meta-model slightly differently to suite the modeling and analysis requirements of these application domains. The following sections present details of the meta-models for Automotive and Avionics domain.

2.2.1 Meta-model for Automotive

The embedded control software development for Automotive applications usually starts from control design. The whole system control, including both the controller with its software and physical system with controlled objects, are specified in some control design and simulation environment. Such control design ensures the correctness of the whole system both functionally and timely. One such control design tool used in Automotive industry is Matlab Simulink and Stateflow diagram.

Simulink and Stateflow component model. The models of the controller software in Simulink and Stateflow can be used as reusable components in the software integration and analysis. In order to use these Simulink and Stateflow models, we must import them to the GME environment, which is the graphic modeling environment our timing model and analysis is based on. The meta-model for Simulink and Stateflow are given in Figure 2.5 and 2.6.

Software architecture model. Software models in AIRES are multi-level models. The software model is constructed in a software folder called SWFolder. The software model is called Target. In the Target model, there can exist multiple subsystem models, each of which models a portion of the controller software. Such subsystems are typically partitioned according to functional design. For example, an Engine controller can be partitioned into the subsystems of Air-Fuel Ratio control, Electronic Throttle control, and Spark control. Multiple components are integrated to implement a subsystem. These components can be those modeled and stored in Simulink and Stateflow folders. Components contain ports for communications, which can be either data ports or event ports. These ports are also classified into input and output ports. A legal connection can only happen between a pair of input and output of the same type of ports. To model software component-processor binding, a reference of a CPU component in platform model can be created in either subsystem or software component(s) or both. The CPU reference in a subsystem model is a simplified version of modeling the CPU references in the components in the subsystem, which indicates all components in the subsystem are bound to the referred CPU. If a CPU references is created in a component instance, the component is bound to the referred CPU. We allow a CPU reference of a component to be different from the CPU of the subsystem that contains the component. In such a case, the component will be running on the specified CPU, while other components of the subsystem will be running on the CPU of the subsystem. For those components without CPU reference, they can be freely allocated to any processor. AIRES analysis

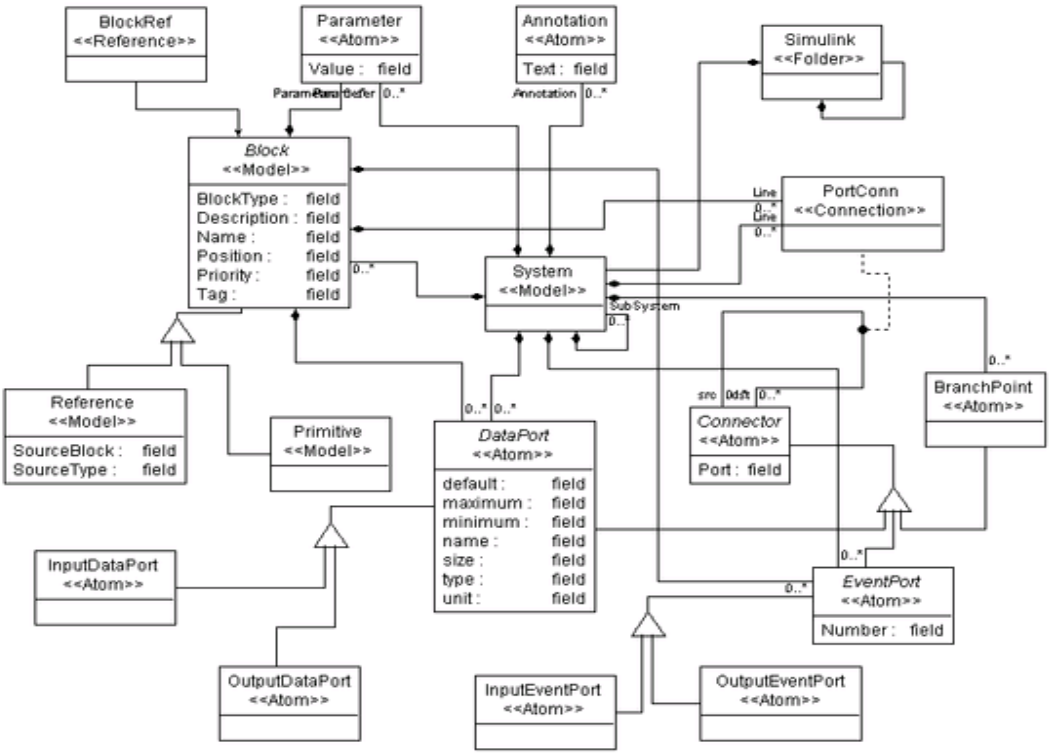


Figure 2.5: Simulink meta-model for Automotive in AIRES.

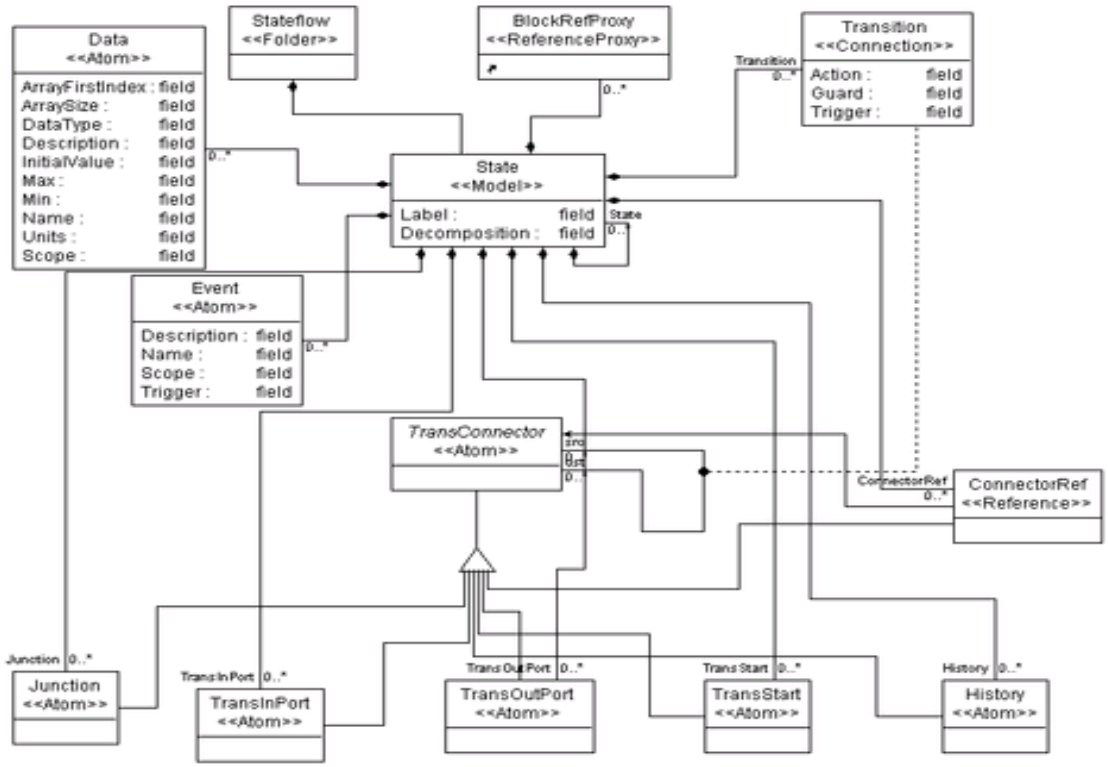


Figure 2.6: Statflow meta-model for Automotive in AIRES.

algorithm can determine the execution location for these components according to selected allocation strategy.

The meta-model also specifies the performance information. The performance requirements are given at the subsystem level, including system period and deadline. Software components have worst-case execution times and priorities associated to them. The data connections also have data size associated with them for deriving the communication delays.

The meta-model of AIRES software model is presented in Figure 2.7.

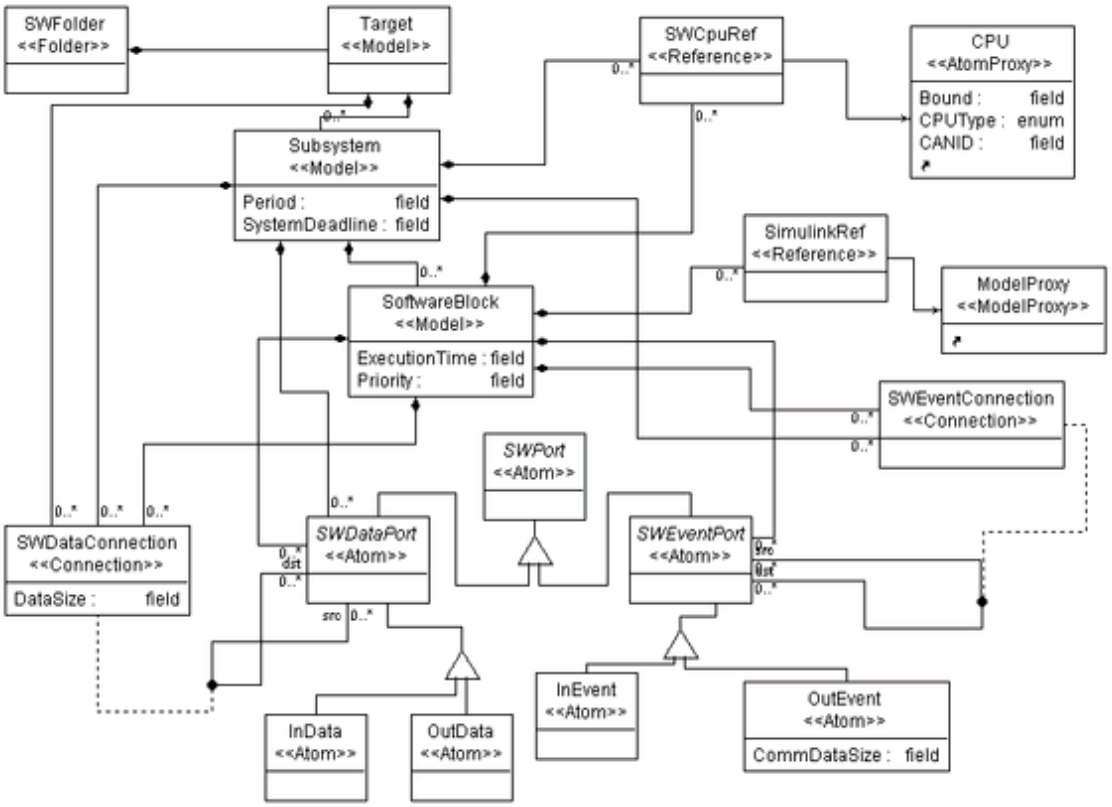


Figure 2.7: Software meta-model for Automotive in AIRES.

Platform configuration model. The platform model in the HWFolder specifies the configuration of the designed software execution environment, including processors (CPU), networks, and operating systems, as shown in Figure 2.8. Performance related parameters such as CPU bound and network connection speed are defined in the meta-model for hardware components. For operating systems, allowed performance parameters include various overheads, such as timer overhead, context switch overhead, and scheduling overhead. Note that the values of these overheads depend on platform configuration such as the type of the processor, the implementation of the OS, and the specified timer resolution. Since we focus on computation-centric applications, and there are usually only a small number of candidate hardware-OS combinations, we argue that the value of these performance related parameters can be measured.

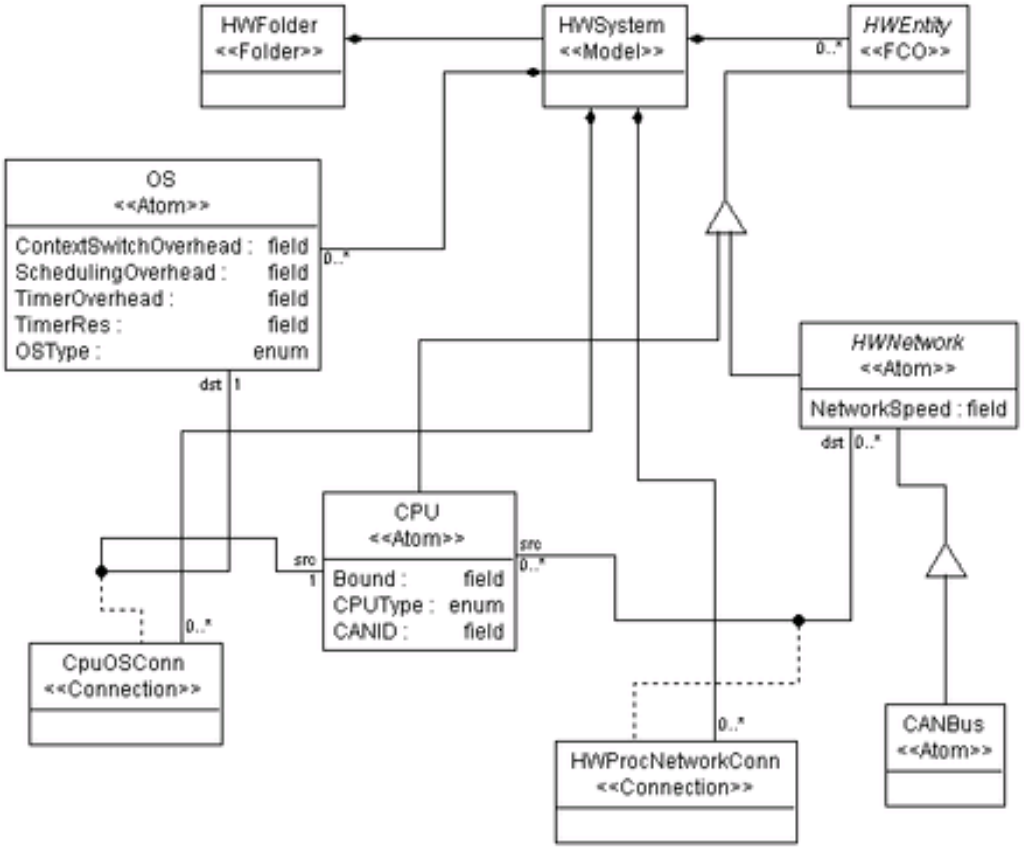


Figure 2.8: Platform meta-model for Automotive in AIRES.

Runtime model. A runtime system of a designed software model consists of a set of tasks. Similar to the software model, a runtime model, called a task system, is stored in the TaskFolder. A task system is also a multi-level model. At the highest level, a task system consists of one or more logical task, each of which defines a task chain. A task in a logical task is the basic schedulable execution unit of the system. It can be implemented as a process/thread of the target operating system. A task consists of a set of actions, each of which is a software component in the software model. As the actions in a task are executed sequentially with invocation dependencies, the communication cost between actions can be negligible. The dependencies between tasks can be either event dependencies or data dependencies. The event dependencies models the control flow, and can be implemented as event trigger like those in QNX, VxWorks, and OSEKWorks. Data dependencies, on the other hand, model the data flow, and can be implemented as inter-process communications. To model these dependencies, a task contains input ports and output ports for either event communication (control flow) or data communication (data flow). Same as in the software model, an input port is only allowed to connect to an output port of the same type. Note that the logical tasks in a runtime system and the subsystems in the software model do not necessarily have an one-to-one mapping. As a subsystem may involve multiple processors, it can be implemented as multiple tasks running on different processors in the runtime model. On the other hand, a logical task typically consists of tasks of the same subsystem, as the tasks of different subsystems may not have any dependency or communication. Since many tasks in ECSW are periodic,

we created a special trigger in the meta-model to model this type of time-triggered task chains. In an application model, the trigger object, called timer, connects only to the event port of the first task in the logical task system. Each task also contains a CPU reference for execution location modeling, similar as the software model.

The performance modeling for a runtime system includes end-to-end timing constraints for a task chain, the period and relative deadline of a task, execution rate and deadline of a logical task chain, and execution time of each task and action. Some of the information defined in this meta-model are derived from the others. For example, the execution time of a task can be computed by adding up the execution times of all its actions. The period and deadline can be derived from the period and deadline of the logical task using some algorithm like deadline distribution [14]. Figure 2.9 shows the meta-model of the AIRES runtime system.

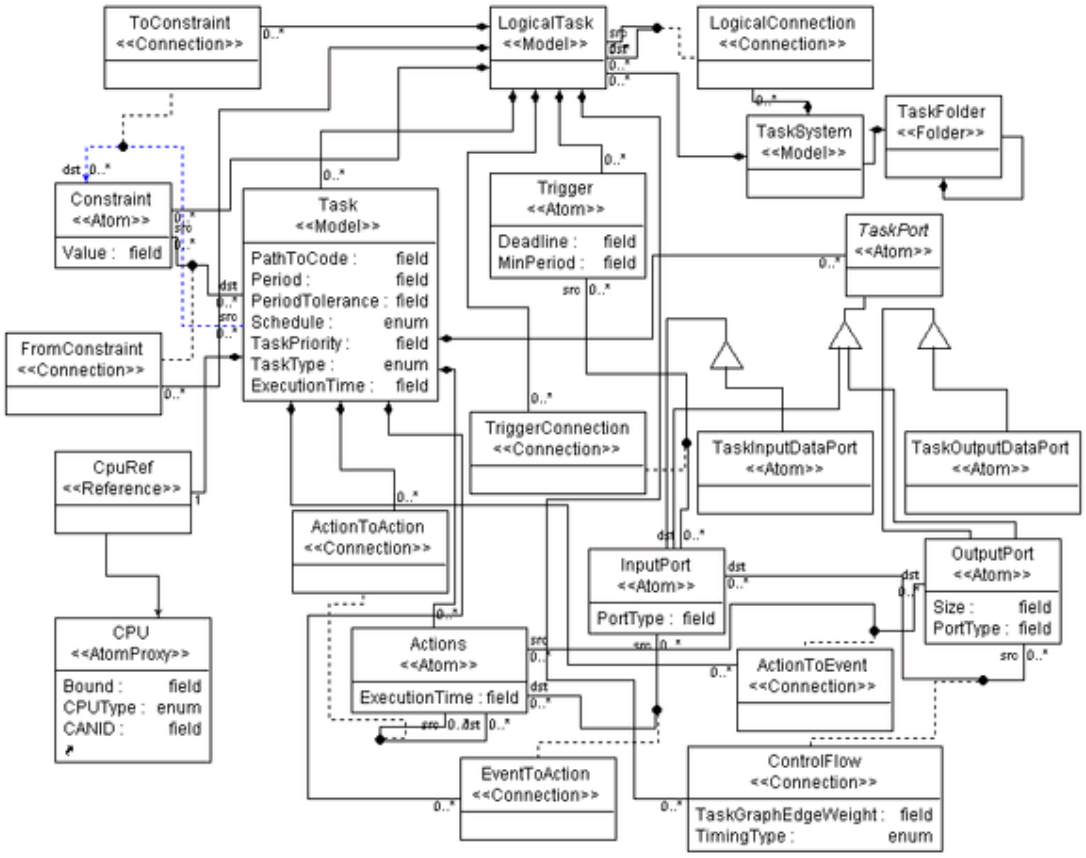


Figure 2.9: Runtime system meta-model for Automotive in AIRES.

2.2.2 Meta-model for Avionics

The meta-model for Avionics applications, called Embedded System Modeling Language (ESML), is developed by Vanderbilt University. It is designed and implemented to model the software in the Bold Stroke framework. The Bold Stroke framework [24, 25] is a product-line architecture used at Boeing for developing avionics mission computing software, which is the embedded software aboard a millitary aircraft for controlling mission-critical functions, such as navigation, target tracking and identification, and weapon firing. The system and components in Bold Stroke framework are modeled in UML [28], manually implemented in C++, and runs on top of Real-Time CORBA Event Service. The initial

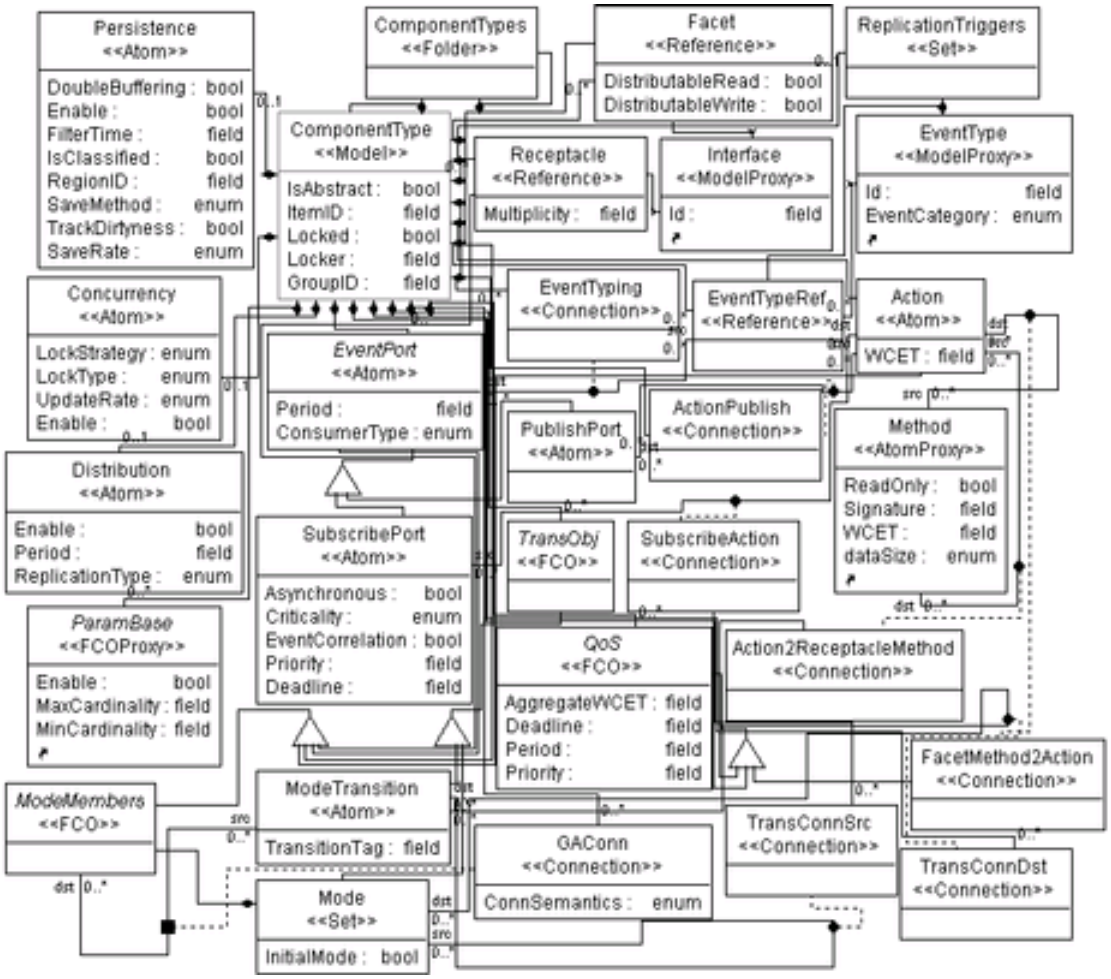


Figure 2.10: Meta-model of Avionics components.

models of components are constructed in Rational Rose. The ESML, therefore, is based on Real-Time Event Channel implemented in the TAO CORBRA [23], and captures all apsects of the embedded software, including software architecture, timing and resource constraints, execution threads, platform information, allocation of components, etc. The models essential for performance analysis in ESML are component models that describes the function and structure of the reusable components, interaction models that model the software architecture, and configuration models that model the runtime system including OS scheduling units and hardware organizations.

ESML component model. The components in ESML are composite objects with ports, which interact with one another, either through event triggers or procedure invocation. Figure 2.10 shows the meta-model of components in ESML.

In ESML, each component can have publish ports to publish events, subscribe ports to receive events, receptacles to issue method invocations, and facets to accept method invocations. Each input port, either a subscribe port or a facet, has an associated action that in turn triggers one or more output ports, either publish ports or receptacles, of the same component. Given the fact that different ports can be associated with different actions, the component model in ESML is data-centric instead of

process-centric. To model the components' performance, a worst-case execution time (WCET) attribute is annotated to every action. There is also a QoS object defined in ESML that can be associated with components, event channels, and invocations to specify the execution times, deadlines, and periods of these entities. For each event port (publish or subscribe), a period of event arrivals can be defined. For a subscribe port, the deadline of an event can be further specified.

ESML interaction model. The software model in ESML is defined as an interaction model. The components in an interaction model interact with each other through their ports. Interactions with events are only allowed between pairs of publish ports and subscribe ports, while interaction with invocations are only allowed between pairs of receptacles and facets. Figure 2.11 shows the meta-model of interaction models.

In this meta-model, component interactions follow a control-push data-pull style. First, the data producer component publishes a DataAvailable event from its publish port, indicating that it has fresh data; when the data consumer component receives the event from its subscribe port, it issues a GetData call from its receptacle to the producer's facet to retrieve the data. Each subscribe port can subscribe to multiple events, and has a correlation attribute, either AND or OR. For AND correlation, the action associated with the port is executed only when all of the input events arrive; for OR correlation, the action associated with the port is triggered when any of the input events arrive. With such an interaction model, we can trace the interactions among the components following their events/invocations to form a directed acyclic graph, called Port Dependency Graph, for performance analysis.

ESML configuration model. The configuration model in ESML models the runtime view of the system. A configuration model contains hardware and network models, the execution process and threads of operating system/middleware, and the location (processor-thread) of software components. The meta-model for configuration is shown in Figure 2.12.

Models in AIF format. ESML provides a superset of information needed for analysis. To reduce the information an analysis tool like AIRES needs to manipulate, we extract analysis related information from an ESML model in the form of Analysis Interface Format (AIF). The AIF is implemented as an XML file, and contains subset of ESML language of dependency and real-time information. With AIF, a third party modeling and analysis tool can be easily integrated in the development tool chain. AIRES tools are designed and implemented to work with AIF. The meta-models of AIF are shown in Figure 2.13 and 2.14.

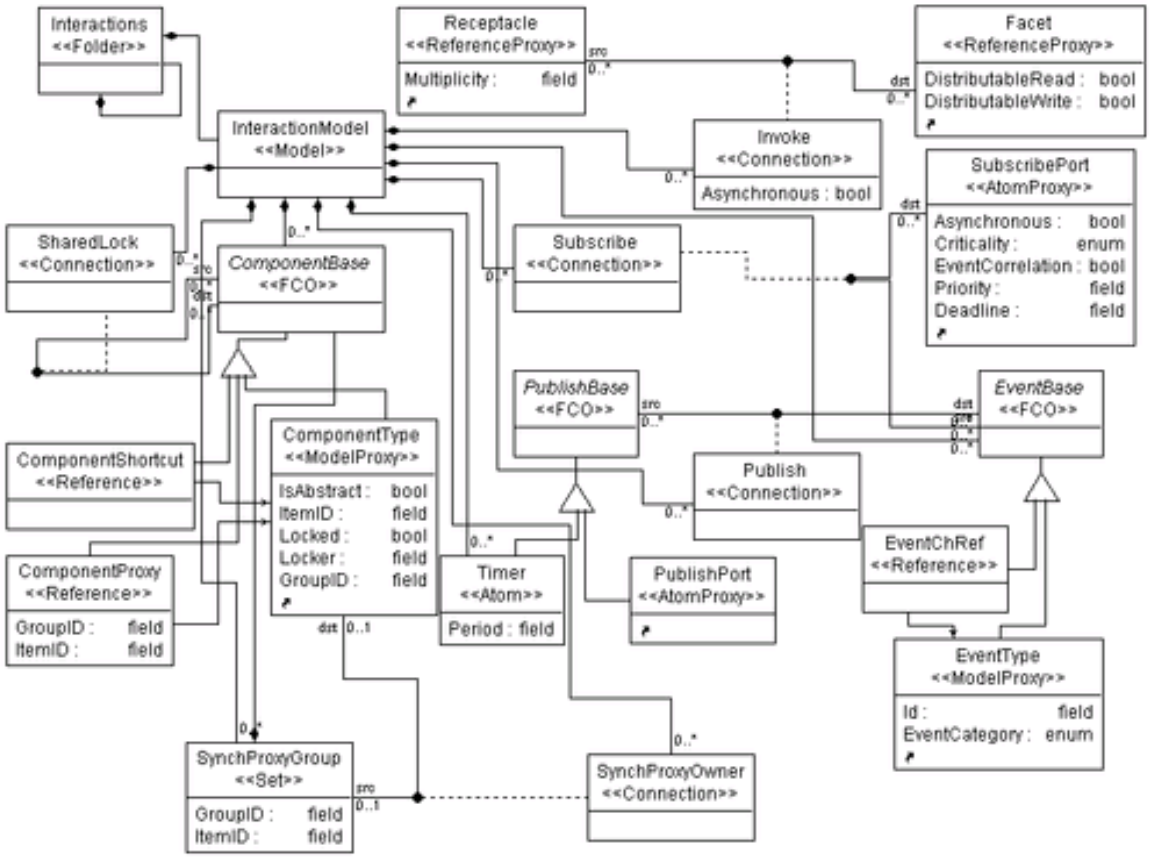


Figure 2.11: Meta-model for Avionics software architecture.

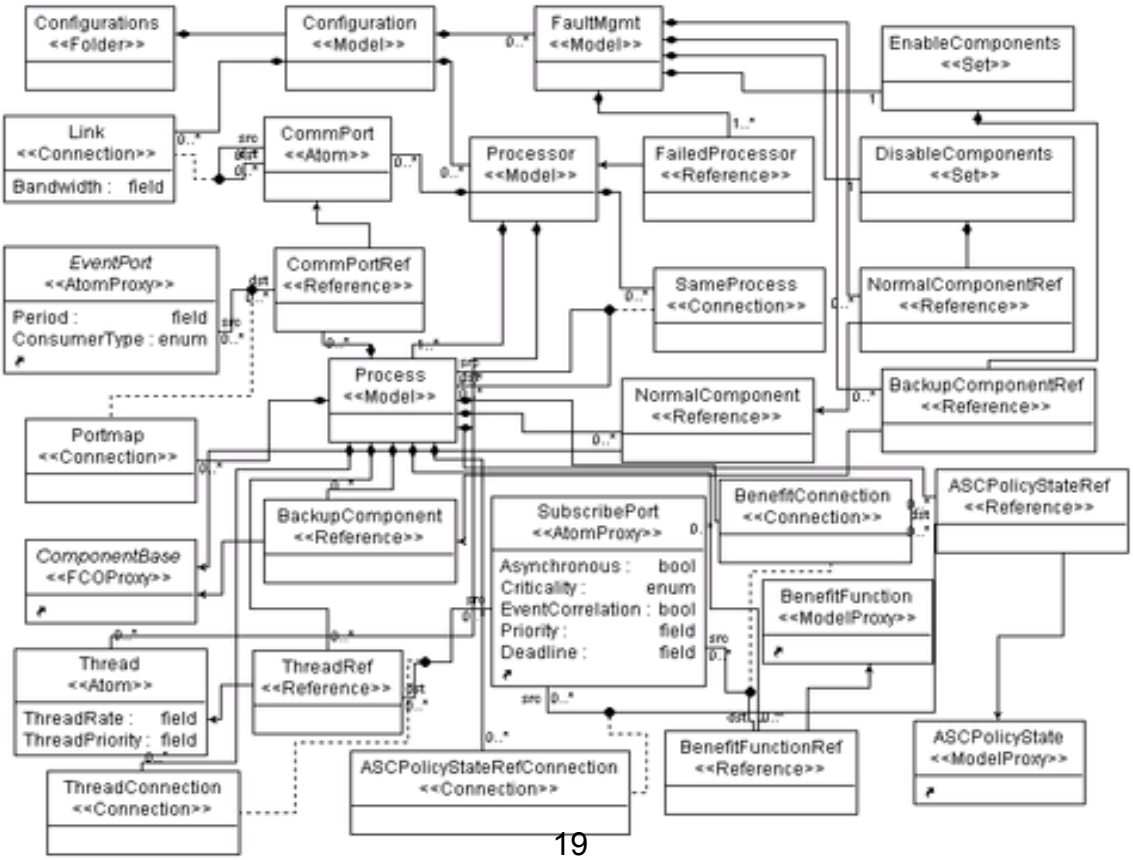


Figure 2.12: Meta-model of Avionics configuration.

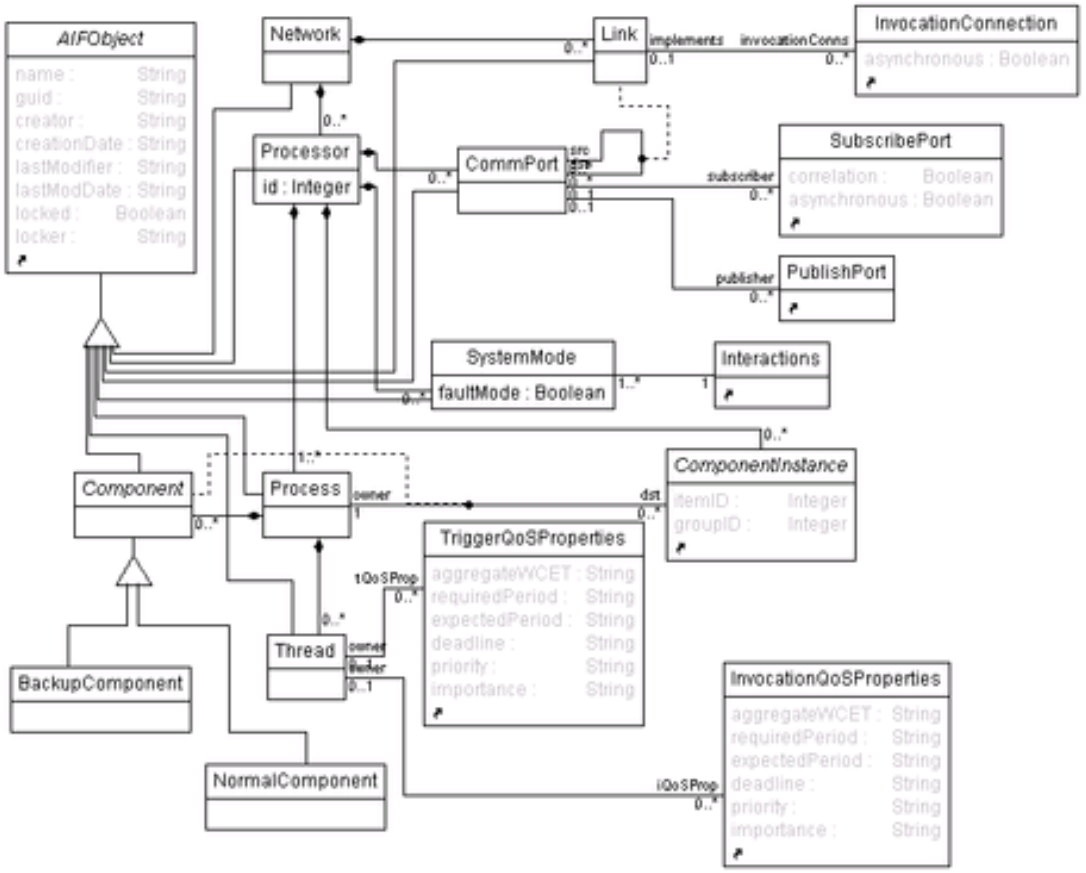
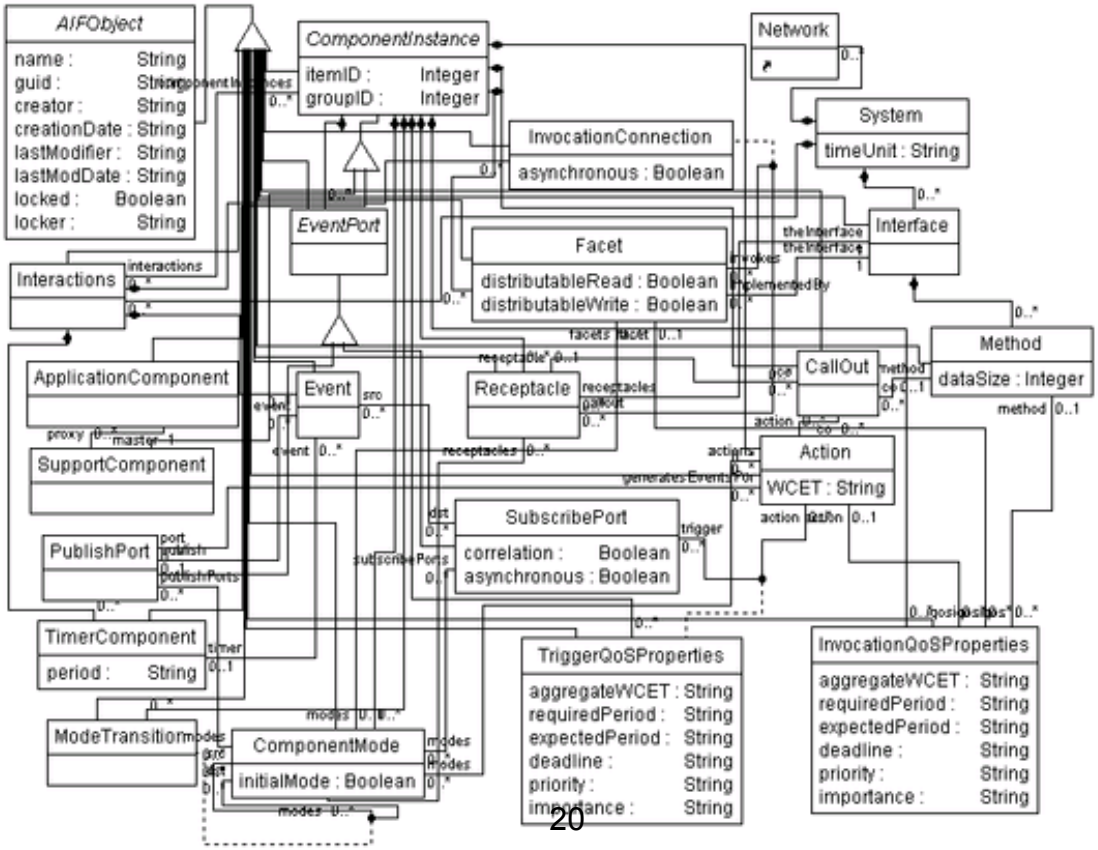


Figure 2.13: Meta-model of configuration in AIF.



Chapter 3

Analysis and Verification Algorithms

The core of the AIRES tool is a set of analysis algorithms manipulating the application models constructed using the meta-model. Some of these algorithms are used to provide design transformation and design recommendation, while the others are use to verify the correctness of the design functions and/or performance.

3.1 Functional composition and check

The functional check algorithms are designed to detect the mistakes made by the designer during the model construction. Since the software model usually consists of several hundred components, it is difficult for the designer to walk through them one-by-one to check whether there exist errors such as wiring ports with different types or creating a cyclic dependencies among components. Such design mistakes are common especially when multiple designers are involved. Our algorithms are designed to automaticaly detect such errors in a given software model so that the designer can easily find the components/events/invocations involved in the errors, and fix them quickly. As the modeling and analysis requirements for Automotive and Avionics applications are different, our algorithm performs the component composition check for Automotive models, and performs event dependency check for Avionices.

Component composition check. Component composition check verifies the correctness of links between components' ports. A correct link must be one of the following:

- The link connects an output port and an input port of different components at the same modeling hierarchy, with the output port as the source and the input port as the destination, as link $D \rightarrow E$ shown in Figure 3.1. The outputs of the outport must also be compatible with the inputs of the input port.
- The link connects a pair of input ports at two immediate adjacent modeling hierarchies, with the port at the higher modeling level as the source and the port at the lower level as the destination, as link $A \rightarrow C$ and $B \rightarrow E$ shown in Figure 3.1. The type of the input port of the higher level inherites the type of the input port of the lower level.
- The link connects a pair of output ports at two immediate adjacent modeling hierarchies, with the port at the lower level as the source and the port at the higher level as the destination, as link $D \rightarrow G$ and $F \rightarrow H$ shown in Figure 3.1. The type of the output port of the higher level inherites the type of the output port of the lower level.

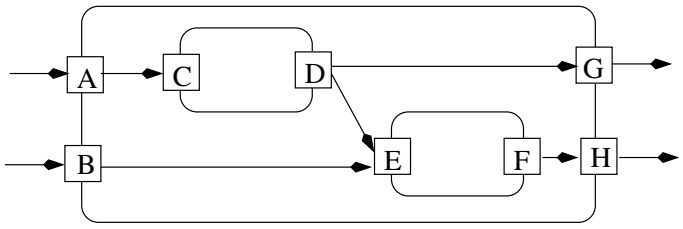


Figure 3.1: Correct links in software model.

Two compatible ports must satisfy the following conditions:

1. The two ports are the same type, meaning they are both either data ports or event ports.
2. If they are event ports, the event at the source port is acceptable for the destination port.
3. If they are data ports, the data at the source and destination are (i) both of the same data type, and (ii) the value range of the source should be equal to or smaller than the value range of the destination.

The component composition check algorithm requires the existence of reusable components. In AIRES toolkit, the reusable components can be either manually created in the software folder, or imported from Simulink/Stateflow diagram. The attributes of the reusable components' ports must be described in a spreadsheet. The required attributes include port name, port type, data type, data length, data dimension, value range, and default value.

The algorithm performs the composition check by traversing the graph of the software model, and comparing the values of linked ports. Algorithm 1 shows the algorithm for the composition check.

Algorithm 1 Algorithm for composition check.

INPUT: software model in graph $M =< C, L >$;
port attributes and values $< p, a, v >$;
OUTPUT: incompatible port set S;
BEGIN
1 $S \leftarrow$;
2 **foreach** $l(s,d) \in L$ **do**
3 $c_s = find_{component}(s)$;
4 $c_d = find_{component}(d)$;
5 $< s, a, v >= find_{attribute}(c_s)$;
6 $< d, a, v >= find_{attribute}(c_s)$;
7 **if** $incompatible(< s, a, v >, < d, a, v >)$ **then**
8 list $\leftarrow (s, d)$;
9 **end-for**
10 **return** S;
END

Event dependency check. The goal of the event dependency check is to detect the existence of event/invocation cycles, and events published without a subscriber or events subscribed with a publisher.

As discussed in previous section, the software model for Avionics is a *data-centric* component diagram. This diagram must be translated into a *port dependency graph* (PDG) for analysis. The design and implementation of the event dependency check in AIRES is based on PDG. In this work, we use ports to refer to both event ports (publish and subscribe) and invocation interfaces (receptacle and facet).

The PDG captures all the relevant dependency information in the ESML model, and serves as the backbone data structure for all subsequent analysis tasks. However, we define two other types of dependency graphs for purposes of convenient visual display as well as easy manipulation in certain analysis tasks. They capture dependency information at a higher level of abstraction – component-level instead of port-level – hiding all the intra-component dependencies. They can be derived directly and straightforwardly from PDG.

Definition 2 *A* Port Dpenedency Graph *(PDG) is a graph* (V_p, E_p)*, where*

- V_p *is a set of ports,* $\{p_i, 1 \leq i \leq N_p\}$*. Each* p_i *can be one of 4 types: publish port* p_{pub}*, subscribe port* p_{sub}*, receptacle* p_{recep}*, or facet* p_{facet}*.*
- E_p *is a set of directed, weighted* port connections, $conn_i, 1 \leq i \leq N_{conn}$*, and each* $conn_i$ *can be one of 2 types:*
  - Inter-component dependency: *is either event-trigger dependency between a* p_{pub} *and a* p_{sub}*, or invocation dependency between a* p_{recept} *and a* p_{facet}*.*
  - Intra-component dependency: *describes the intra-component trigger pathways from input ports to output ports of the same component.*

The weight of an edge is equal to the execution rate assigned to the ports that it connects. [1]

Definition 3 *A* Component Event Dependency Graph (CEDG) *is a graph* (V_c, E_{ce})*, where*

- V_c *is a set of components,* $\{c_i, 1 \leq i \leq N_c\}$*.*
- E_{ce} *is a set of directed, weighted component event connections, which are derived from inter-component connections between event ports on the components.*

Definition 4 *A* Component Invocation Dependency Graph (CIDG) *is a graph* (V_c, E_{ci})*, where*

- V_c *is a set of components,* $\{c_i, 1 \leq i \leq N_c\}$*.*
- E_{ci} *is a set of directed, weighted component invocation connections, which are derived from the inter-component connections between invocation ports (receptacles and facets) on the components.*

Due to the control-push data-pull interaction style, CIDG is in most part a *reverse graph* of CEDG, i.e., a graph obtained by reversing the direction of the edges of CEDG, but it's also common to find event connections without corresponding invocation connections, or vice versa. Similar analysis techniques can be applied to both types of graphs. We mostly focus on event dependencies in this paper.

We can use conventional graph algorithms to analyze the dependency graphs, and identify certain anomalies such as:

- Dependency cycles. A cycle of event or invocation dependencies usually indicates a design error since it becomes an infinite loop at runtime.

[1] The edge weight cannot be assigned until the rate assignment algorithm has been run.

- Events published with no subscribers, or events subscribed to with no publishers. AIRES provides warnings to the designer, even though this may not necessarily be an error.
- Component ports unreachable from any timers, hence unable to be assigned rates. This is elaborated in Section 3.2.2.

We can also perform *forward/backward slicing* of the dependency graphs. Given a component or a port, we can answer user queries such as

- What downstream components/ports can this component or port potentially affect via event or invocation dependencies?
- What upstream components/ports can potentially affect this component or port?

This is achieved by traversing the dependency graphs forward or backward starting from a component (for CEDG and CIDG) or a port (for PDG). These queries are useful in software evolution, where a designer can assess the impact of changing or replacing a certain component, as well as for other purposes such as localizing faults, minimizing regression tests, reusing components, and system re-engineering.

Even though the current avionics software does not allow dynamic creation or destruction of components, both the inter- and intra-component dependencies can change at runtime due to *modal* behavior, that is, components can change mode to publish new events, stop publishing old events, or change its internal trigger pathways. For example, a modal component can have both active and inactive modes. When in the active mode, an input event triggers an output event; when in the inactive mode, an incoming event is simply ignored and dropped. ESML allows modeling of such behavior by associating a finite state machine with a modal component. Instead of a single PDG, we can view the system as having multiple pre-defined system-level modes, obtained by all combinations of component modes. We can construct a PDG for each system-level model, and apply the analysis techniques to each mode separately.

3.2 Runtime model generation

The runtime model generation is a process of allocating the components in the software model to the processors in the platform model, assigning the timing attributes for these components and communications for execution schedule, and forming tasks that are individually schedulable by the operating systems in the processors.

3.2.1 Component allocation

Since the embedded software usually runs on a resource-limited platform, we considered allocation strategies that lead to minimize the resource usage. Specifically, we considered computation resources (CPUs) and communication resources (networks) in this work. The implemented strategies include *first-fit*, *load-balance*, *communication-minimization*, and the combination of *communication-minization* and *first-fit* or *load-balance*.

All of the implemented strategies require knowing the resource demands of components and communications. We used the utilizations of the software components to model their computation resource demands. and used the message size passed between ports to model the communication resource demands. The utilization of a software component can be computed as:

$$U_c = \frac{e_c}{P_c}$$

where U_c is the utilization of the component c, e_c is the worst-case execution time (WCET) of c, and P_c is the invocation period of c. The invocation period P_c must be derived for every component c in the software model from high-level system rate specifications. This will be further detailed in Section 3.2.2.

First-fit allocation algorithm. The first-fit allocation strategy aims at minimizing the total number of the processors in the system. In the first-fit allocation algorithm, the current processor must be fully utilized where no more component can be further allocated to it before the algorithm moves to another processor. The algorithm is shown in Algorithm 2.

Algorithm 2 Algorithm for first-fit allocation.

INPUT: software model in graph $M =< C, L >$;
utilization U_c for each $c \in C$;
a set of processors $P = \{P_i\}$ with utilization bounds $\{B_i\}$.
OUTPUT: components on processors $C(P_i)$.
BEGIN
1 sort $c \in C$ so that $U_1 \geq U_2 \geq \cdot \geq U_n$;
2 sort P_i so that $B_1 \geq B_2 \geq \cdot \geq B_m$;
3 **for** P_1 to P_m **do** $U_r = B_m$;
4
5 **while** $(P \neq \phi) \wedge (C \neq \phi)$ **do**
6 $P_f leftarrow first(P)$; $P \leftarrow P - \{P_f\}$;
7 $U_f \leftarrow B_f$;
8 $c \leftarrow first(C)$;
9 **for** every $c \in C$ **do**
10 **if** $U_f \geq U_c$ **then**
11 $C(P_f) \leftarrow CP_f \cup \{c\}$;
12 $U_f \leftarrow U_f - U_c$;
13 $C \leftarrow C - \{c\}$; $c \leftarrow first(C)$;
14 **end-if**;
15 **end-for**;
16 **end-while**;
17
18 **if** $C \neq \phi$ **then return** fail, not enough processors;
19 **else return** $\{C(P_i)\}$;
END

In Algorithm 2, the execution locations of the components depend solely on their computation resource demands, and the communications are all ignored. The algorithm starts from the processor with the most capacity, tries to allocate components one-by-one if the component can fit in the current processor, and removes the component from the unallocated set to the current processor set. In the case that there is no component that can be allocated to the current processor, the algorithm picks another one in the processor list. In the implementation of this algorithm, we took the manually allocated components (for processor-bound components) on a processor into account by substracting the resource consumed by the pre-allocated components for the utilization bound of the processor. This is due to the fact that the first-fit algorithm tries to fully utilize the processor, therefore allocating a component to the processor is the same as lowering the available resource for further allocations. The algorithm terminates when there are no more processors (fail), or all components are allocated. The computation complexity of the first-fit algorithm is $O(nm)$, if there are n software components, and m processors. Although the first-fit algorithm yields an allocation that requires the minimum number of computation resources, the final communication resource consumption may not be optimal.

Load-balance allocation algorithm. The load-balance allocation algorithm aims to distribute the workload evenly among all processors in the platform. Although the first-fit allocation requires the minimum number of processors, all processors except the last one are fully used, and can not ensure meeting the timing constraints when the workloads changes, such as transient overload. On the other hand, the platform is usually designed according to other constraints such as reliability and device location. For example, the software for engine control, transmission control, and entertainment device control should be allocated on different processors in an automotive system. In this case, at least 3 processors are in the platform. We, therefore, would like to evenly distribute the workloads of the software across all processors. Doing so provides each processor with some free resource to handle the dynamic workloads that may not be predictable in the design. The detailed algorithm is given in Algorithm 3.

Algorithm 3 Algorithm for load-balance allocation.

INPUT: software model in graph $M =< C, L >$;
utilization U_c for each $c \in C$;
a set of processors $P = \{P_i\}$ with utilization bounds $\{B_i\}$.
OUTPUT: components on processors $C(P_i)$.
BEGIN
1 sort C according to U_c in descending order;
2
3 **while** $C \neq \phi$ **do**
4 **for** $P_c \in P$ **do**
5 $U(P_c) \leftarrow \sum_{c \in C(P_c)} U_c$;
6 **end-for**;
7 sort P according to $U(P_c)$ in ascending order;
8 $P_f \leftarrow first(P)$;
9 $c \leftarrow first(C)$;
10 $C \leftarrow C - \{c\}$;
11 $C(P_f) \leftarrow C(P_f) \cup \{c\}$;
12 $U(P_f) \leftarrow U(P_f) + U_c$;
13 **end-while**;
14
15 **if** any $U(P_c) > B_c$ **then return** fail, not enough resource;
16 **else return** $C(P_i)$;
END.

Algorithm 3 allocates the component with the most resource demands to a processor with the most available resources. So we need to compute the total utilization of every processor to determine which one has the least workload (potentially the most available resource). This computation must be repeated every time a component is allocated, since the new allocation changes the utilization of a processor, and may consequently alter the processor for next allocation. The algorithm terminates when all components are allocated. If there exist any processor whose utilization is greater then its utilization bound, it indicates that there is not sufficient resource to run all the components. Otherwise, $C(P_i)$ contains components should be allocated on each processor. The computation complexity of Algorithm 3 is $O(nlogm)$, where n is the number of software components, and m is the number of processors. Note resorting P in the algorithm after a component is allocated is only a process to find a proper position

in the sorted list followed by an insertion of P_f. This takes $O(logm)$ time. In the implementation, we sorted P first before entering the **while** loop, and using a binary search to resort P.

Communication-minimization allocation algorithm. Communication-minimization algorithm aims to minimize the communications among a set of processors. As more embedded systems are distributed, the communication delays become critical in satisfying the system performance constraints since such delays usually are much larger when compared with computation delays. Meanwhile, most of current targets tend to use shared communication links to connect multiple processors in order to save costs, and passing fewer messages over the shared link reduces the collisions which can, therefore, potentially improve the system performance.

Finding a min-cut of a graph is known to be NP-hard. Therefore, we used a heuristics developed in [1] to find a sub-optimal solution. The heuristics uses the sum of the cost of all incident edges of a node as the weight of the node. As the communication cost after a cut is the sum of the the costs on all edges, the node cost used in the heuristics represents the total communication cost after a cut. Therefore, finding a cut that minimizes the communication becomes finding a cut that minimizes the node cost. Algorithm 4 presents the algorithm allocating the software to a set of processors with the minimum communication costs.

Algorithm 4 Algorithm for allocation with minimum communications.

INPUT: software model in graph $M =< C, L >$;
utilization U_c for each $c \in C$;
a set of processors $P = \{P_i\}$ with utilization bounds $\{B_i\}$.
OUTPUT: components on processors $C(P_i)$.
BEGIN
1 **foreach** $c \in C$ **do**
2 $w_c \leftarrow r_c * (\sum_{(c,i)\in L} l(c,i) + \sum_{(i,c)\in L} l(i,c))$;
3 $C \leftarrow sort(C, descend)$;
4
5 **for** $n = 1to|C| - |P|$ **do**
6 $c \leftarrow first(S)$;
7 $d \leftarrow node(max\{l(c,i), l(i,c)\})$;
8 $cd \leftarrow \{c, d\}$;
9 $w_{cd} \leftarrow r_c * (\sum_{(c,i)\in L-(c,d)} l(c,i) + \sum_{(i,c)\in L-(d,c)} l(i,c)) + r_d * (\sum_{(d,i)\in L-(d,c)} l(d,i) + \sum_{(i,d)\in L-(c,d)} l(i,d))$;
10 $C \leftarrow sort(S - c, d + cd, descend)$;
11 **end-for**;
12
13 **foreach** $P_i \in P$ **do**
14 $C(P_i) \leftarrow first(S)$;
15 $C \leftarrow C - C(P_i)$;
16 **end-for**;
17
18 **return** $C(P_i)$;
END.

Algorithm 4 first computes the weight of each component, w_c, in the model as the sum of the costs of all its edges, then sorts the components in the descending order according to w_c. As there are $|C|$

number of components allocated to $|P|$ number of processors (assume $|C| \geq |P|$), C must be partitioned into $|P|$ groups. In each step, the algorithm picks one component c with the highest weight, finds the highest cost edges l among all its incident edges, and merges the two components connected to l. Such merge indicates the two components should be allocated to the same processor to yield the minimum communication after the cut. The two components are then considered as one inserted back to the component set. The weight of the new merged components is computed as the sum of the edge costs from/to both components excluding the edge between them. The component set must be resorted after each merge. After each step, $|C|$ is reduce by 1. The algorithm terminates in $|C| - |P|$ steps when $|P|$ number of components left in C. Each component in C at the end of the algorithm represents a group of components allocated to the same processor. Sorting C after each step takes $O(logn)$ time where n is the number of components, and the algorithm needs $(n - m)$ steps to find m partitions, where m is the number of processors. The computation complexity of Algorithm 4 therefore is $O((n - m)logn)$.

Combinational allocation algorithm. The combinational allocation algorithm aims to achieve multiple goals instead of one when allocating the software components. In our work, the goals are from both computation resource constraints and communication resource constraints. Specifically, we considered the combinations of the strategies discussed previously, including the combinations of first-fit and communication-minimization, and load-balance and communication-minimization.

Since different allocation strategies may conflict with each other in nature, we implemented the combinational allocation algorithms with user defined primary consideration and secondary consideration. The primary consideration is used to first to allocate the software components, and the secondary consideration is used to make choices among different allocations generated using the primary consideration and with the same performance. For example, using first-fit with communication-minimization strategy, the primary consideration is first-fit algorithm, and the secondary consideration is communication-minimization. In case that allocating different components yields the same utilization of the a processor, the one with less communication cost is selected.

Figure 3.2 gives a software structural model. The values for nodes are the computation resource demands (in utilization), while the values along the links are the communication resource demands. Suppose the allocation strategy is first-fit with communication-minimization, and there are 3 homogeneous processors with utilization bounds are all 1. The process of the allocation is given in Table 3.1.

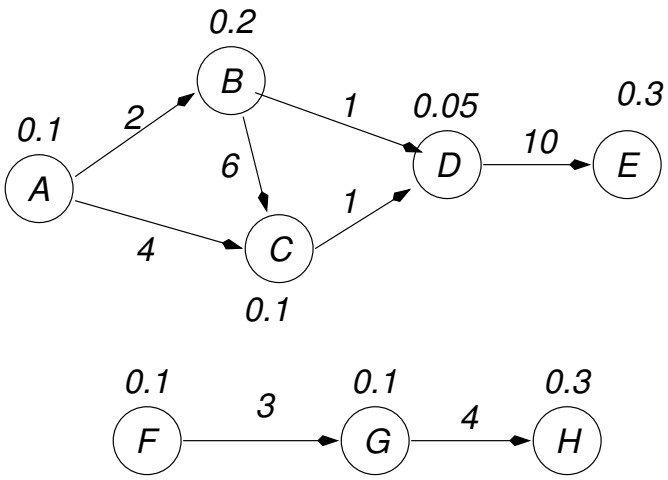


Figure 3.2: Software model of the exmaple.

step	working processor	allocated components	unallocated components
1	P_1	(E)	H, B, F, A, G, C, D
2	P_1	(E, H)	B, F, A, G, C, D
3	P_1	(E, H, B)	F, A, G, C, D
4	P_1	(E, H, B, C)	F, A, G, D
5	P_1	(E, H, B, C, A)	F, G, D
6	P_2	$(E, H, B, C, A)(F)$	G, D
7	P_2	$(E, H, B, C, A)(F, G)$	D
8	P_2	$(E, H, B, C, A)(F, G, D)$	

Table 3.1: Allocation steps.

According to the first-fit allocation algorithm, the result of the example will be component (E, H, B, F, A) on P_1 and (G, C, D) on P_2. Using the combinational allocation of first-fit with minimization-allocation, the component C will be allocated on P_1. This is because allocating C, or F, or A, or G on P_1 yields the same utilization. Therefore, the one that will result in less total communication cost will be chosen, which is C. Note that this is also a heuristic search for a sub-optimal solution in the algorithm, as the optimal solution requires examining all permutation of allocations with the same utilization which is NP-hard. Given the size of computation-intensive embedded software with a large number of components but small number of processors, these heuristics are essential for scalability. The same strategy is applicable for other combinations (load-balance with minimization-communication, minimization-communication with first-fit or load-balance).

The algorithm is illustrated in Algorithm 5. The algorithm first allocates the components according to the user-defined primary consideration. If there are multiple choices leading to the same results using the primary consideration, it relies on a tie-breaking function, *ResolvTie()*, to choose one among the alternatives using the secondary consideration. The *ResolvTie()* function takes the unallocated components, checks those combinations leading to the same results for the primary consideration, and chooses the one meeting the secondary consideration.

Note that finding an equivalent alternative allocations for a component c with arbitary combination of other components is also NP-hard (map to integer knapsack problem). Therefore, we used a heuristics for the selection of the alternatives. When the secondary consideration is first-fit or load-balance, we used the componet's utilization as the heuristics, and chose the one with the maximum utilization first. When the secondary consideration is communication, we used the number of connected components as the heuristics, and chose the one with the most connected components.

With all the heuristics given above, the computation complexisty of the combinational allocation algorithm is $O(n^3)$, where n is the number of the software components in the model.

3.2.2 Timing attributes assignments

The runtime model of a designed ECSW must include timing attributes for each components in order to construct a schedule for runtime execution. However, the timing specifications given in the behavior and structural models is usually expressed as high-level end-to-end constraints, such as execution rates and deadlines of the system transactions. These high-level constraints must be partitioned and assigned to the constitute components for scheduling. In this work, we applied two techniques to derive timing attributes for the software components in the structural model, namely *rate propogation* and *deadline distribution*.

Algorithm 5 Algorithm for combinational allocation.

INPUT: software model in graph $M =< C, L >$;
utilization U_c for each $c \in C$;
a set of processors $P = \{P_i\}$ with utilization bounds $\{B_i\}$;
considerations Pr and Se.
OUTPUT: components on processors $C(P_i)$.
BEGIN
1 **foreach** $c \in C$ **do**
2 allocate c to P_i according to Pr;
3 **if** $(\exists c_a \subset C) \wedge (C(P_i) \leftarrow \{c_a\} \stackrel{Pr}{\simeq} C(P_i) \leftarrow cunderPr$ **then**
4 $c_s \leftarrow ResolvTie(P_i, C, c)$;
5 $C(P_i) \leftarrow c_s$;
6 $C \leftarrow C - \{c_s\}$;
7 **else**
8 $C(P_i) \leftarrow c$;
9 $C \leftarrow C - \{c\}$;
10 **end-if-else**;
11 **end-for**;
12
13 **if** $C = \emptyset$ **then return** $C(P_i)$;
14 **else return** $Fail, notenoughresource$;
END.

ResolvTie(P_i, C, c)
15 **BEGIN**
16 **foreach** $c_a such that (C(P_i) \leftarrow c_a) \stackrel{Pr}{\simeq} (C(P_i) \leftarrow c) underPr$ **do**
17 $cost(c_a) \leftarrow costofSeforallocating c_a to P_i$;
18
19 $c_s \leftarrow \{c_a : min(cost(c_a)\}$;
20 **return** c_s;
21 **END**.

Rate propogation. The rate propogation is used to assign invocation rates of the components in a model. Knowing the execution rate of a component is essential for computing the component's resource demands, which are consequently used in the component allocation algorithms to determine the execution location of the component.

In an embedded system, the software execution is usually triggered by the stimuli, in the form of either data arrivals or event occurances, from the controlled external world. From the software perspective, this indicates the invocation of input components, such as periodic sensor reading or invocation of interrupt handler. The arrivals of input data/events at the input components will then trigger a chain of executions of other components to process the information until the control commands are generated and sent out by the output components. In the system specification, the execution rate is typically given as the rate of an input or output. Therefore, by following the information processing flow from an input forward (or an output backward), we can derive the rate of all components in a transaction.

An issue in such an approach is that there exist some components shared by multiple transactions

running at different rates. A shared component can either be invoked upon the arrivals of all the inputs, called an AND invocation, or the arrival of any input, called an OR invocation. For an AND invocation, the component's rate should be the lowest (or the highest for some application) rate of the component's predecessors, while the rate should be the sum of all its predecessors for an OR invocation. To simplify the analysis, an OR invocation can be modeled as duplications of the component in every transaction sharing it. Therefore, we only considered the AND invocations in the rate propogation algorithm. The rate assignment algorithm is shown in Algorithm 6.

Algorithm 6 Algorithm for rate assignment.

INPUT: software model in graph $M =< C, L >$;
rate specifications for each input R;
OUTPUT: component C with rate assigned.
BEGIN
1 **foreach** $c \in C$ **do** $c.color = WHITE$;
2 $R \leftarrow sort(R, ascend)$;
3
4 **foreach** $r \in R$ **do** dfs_search(c_r, r);
5
6 **return** C;
7 **END**.

dfs_search(c_r, r)
8 **BEGIN**
9 $c_r.color \leftarrow GRAY$;
10 $c_r.rate \leftarrow r$;
11
12 **foreach** $c \in \{immediatesucessorof\, c_r\}$ **do**
13 **if** $c.color = WHITE$ **then** dfs_search(c, r);
14 $c_r.color \leftarrow BLACK$;
15
END.

Algorithm 6 first sorts the input components in ascending order according to their rates, and colors all components in WHITE indicating not visited. From each input component with the highest rate in current set, the algorithm colors the component in GRAY to indicate an initial assignment, and assigns the rate as the input rate of the transaction. The algorithm then tracks forward every component reachable from the input component, and assign its rate the same as its predecessor. If the visisted component's color is other than WHITE, the component has already been assigned a rate. The algorithm then tracks backward and finalizes the color in BLACK. As the algorithm starts from the lowest rate and returns when a component has been assigned a rate, a component shared by multiple transactions with different rates is assigned the lowest rate. In the case that the component should be invoked upon arrivals of the highest rate input, the input components should be sorted in descending order of their rates. The computation complexity of Algorithm 6 is $O(n + l)$, where n is the number of components, and l is the number of links.

Deadline distribution. The deadline distribution algorithm is used to determine the release offset and relative deadline of the components within a transaction. The components in a transaction are usually subject to precedent constraints, and the release times and deadlines of these components in sequence should not be overlapped. Exploring such information can greatly help to determine an effective and efficient schedule of the components' executions.

The deadline distribution is performed after components' allocations have been determined. After this, whether the communication between two components is remote or local has been decided and can be accounted in deadline distribution to obtain realistic values.

In this work, we adopted the basic deadline distribution algorithm in [14]. The algorithm takes a system-level end-to-end deadline of a transaction, and distributes it over its constitute components according to the precedent order of their executions. The system-level end-to-end deadline of a transaction is user-specified and can be derived from the control design. Note the end-to-end deadline may or may not be equal to the transaction's period. To distribute the end-to-end deadline, the algorithm first identifies a critical path, which is a sequential component execution following the precedent constraints. The deadline is first distributed over the critical path. Given n_c components and n_l communications (typically communications between components on different processors with non-zero costs) with execution time e_c for each component, and e_l for each communication on the critical path P, an end-to-end deadline D, the intermmediate release times r_c and deadlines d_c can be computed as follows:

$$\begin{aligned} s &= \frac{D-(\sum_{c\in P} e_c+\sum_{l\in P} e_l)}{n_c+n_l} \\ r_c &= d_{c-1} \\ d_c &= r_c + e_c + s \end{aligned} \tag{3.1}$$

In the above equation, a communication is treated the same as a component. s is defined as the average slack for each component/link. A component/link $c-1$ represents the immediate predecessor of the link/component c on P. The release time of the input component $r_0 = 0$, and the deadline for the output component $d_{n_c} = D$.

After the deadline is distributed on the critical path, the components on other paths can be assigned using the second and third equations in Equation 3.1. The deadline distribution algorithm is given in Algorithm 7.

The deadline distribution algorithm distributes the deadline for each transaction at a time. Suppose there are T transactions, the algorithm performs T loops to distribute all deadlines for all transactions. In each loop, finding a critical path can be done through traversing the transaction graph that takes $O(n_t + l_t)$ time for a transaction with n_t components and l_t links. The loops to assign release times and deadlines for predecessors and successors visits all links in the graph, and takes $O(l_t)$ time. Since total number of components from all transactions is $n = \sum_t n_t$ and total number of links are $l = \sum_t l_t$, the computation complexity of Algorithm 7 can be bound by $O(l(n + l))$.

3.2.3 Task formation

All the models we manipulated in the previous sections are at the component level. At this point, we have allocated the components to their execution processors in the platform, and assigned the timing properties for each oneem, including invocation rate, deadlines, and release times. To execute the components accordingly, the operating system on the processor must determine the schedule of the executions of the components allocated on it. Almost all of today's commercial real-time operating systems schedule the execution of the code in the form of threads/processes. This requires that the components must be mapped to threads/processes to be schedulable by the target operating system. We call this step *task formation*. A task is then a thread/process that can be individually scheduled in the operating system. Although we can implement every component in the model as an individual

Algorithm 7 Algorithm for deadline distribution.

INPUT: software model in graph $M =< C, L >$;
end-to-end deadlines of transactions D_t;
OUTPUT: M with intermediate deadline assigned to C and L.
BEGIN

```
1  foreach transaction t do
2    T ← C + L;
3    P ← find_critical_path(t);
4    s ← compute_slack(M, D_t);
5    foreach c ∈ P do
6      if (c  C) ∨ ((c ∈ L) ∧ (e_c ≠ 0)) then
7        r_c ← d_{c-1};
8        d_c ← r_c + e_c + s;
9      end-if;
10   end-for;
11   T ← T − P;
12
13   foreach c ∈ T do
14     foreach c_p = pre(c) do
15       d_{c_p} ← r_c;
16       T ← T − {c_p};
17     end-foreach;
18     foreach c_s = suc(c) do
19       r_{c_s} ← d_c;
20       T ← T − {c_s};
21     end-foreach;
22   end-foreach;
23 end-foreach;
24
25 return M;
```

END.

task, the resources can be used more effectively and efficiently if multiple components are grouped into one task, since the resource consumptions for the RTOS managing tasks increase significantly as the number of task increases.

The task formation is simply a process of grouping components on the same processor for all processors. Given component i and an existing task τ, i is merged with other components in τ if all of the follows are true:

1. Component i and τ reside on the same processor;
2. Component i runs at the same rate with τ;
3. Component i has either immediate predecessor/sucessor in τ, or has no precedent constraints with any component in τ.

The first condition ensures the task boundary is within a processor so that it can be implemented as

local RTOS thread/process. The second condition minimizes the number of tasks by merging components with the same rate and at the same processor in one task, so consequently reduces the resource consumption for task management. The third condition guarantees that two components in a sequential path stay in different tasks if some components between the two components on the path is running as a different task. For example, given a component chain $a \rightarrow \cdot \rightarrow b \rightarrow \cdot \rightarrow c$, if a is in τ_1 on processor P_1, and b is in τ_2 on P_2, c can not be merged with a in τ_1 even if c is allocated on P_1 and runs at the same rate with a. The algorithm is shown in Algorithm 8.

Algorithm 8 Algorithm for task formation.

INPUT: software model in graph $M =< C, L >$;
component allocation $C(P_i)$;
OUTPUT: task set τ.
BEGIN
1 **foreach** processor P_i **do**
2 **foreach** $c \in C(P_i)$ **do**
3 **if** $\exists \tau_i$ has the same rate of c that satisfies all conditions **then**
4 $\tau_i \leftarrow c$;
5 **else**
6 $\tau_i \leftarrow new\_task()$;
7 $rate_{\tau_i} \leftarrow r_c$;
8 $\tau_i \leftarrow c$;
9 **end-if-else**;
10 $wcet_{\tau_i} \leftarrow wcet_{\tau_i} + w_c$;
11 **end-foreach**;
12 **end-foreach**;
13
14 **foreach** τ_i **do**
15 **if** $\exists (c_i \rightarrow c_j) : (c_i \in \tau_i) \wedge (c_j \in \tau_j)$ **then**
16 $l \leftarrow (\tau_i \rightarrow \tau_j)$;
17 **end-foreach**
18 **return** $< \tau, l >$; **END**.

The algorithm checks the components on each processor. Since the components are partitioned in disjoined sets when allocated to processors, this is equivalent to checking every component once. The condition check in the **if** statement searchs existing tasks (can be more than one task for each rate) to find a matching rate, which takes $logt$ time for t number of tasks sorted by their rates. The condition check also checks if there is immediatie connections with the components in the task, which takes up to link number l. So the computation complexity of the algorithm is $O(n(logt+l))$, where n is the number of components.

After tasks are formed, the transactions in the system are now represented as a task graph instead of a component graph. Given the component graph is directed acyclic graph, the obtained task graph must be a directed acyclic graph using Algorithm 8. The task containing an input component is then called input task of the transaction, while the task containing an output component is called output task.

3.3 Resolving task dependencies

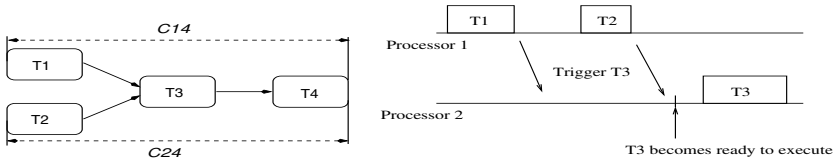
The task dependency resolution was designed to transform a system with dependent tasks into one with independent tasks so that the scalable task allocation and scheduling algorithms can be directly applied. Previous research has indicated that the allocation and scheduling of a system with independent tasks has much lower computation complexity than that of a system with dependent tasks. Therefore, by eliminating the precedent constraints among the tasks, we can obtain a scalable algorithm to allocate and schedule large-scale systems.

Our approach of resolving task dependencies is based on inserting shared buffers between pairs of dependent tasks. Given a task system A with inter-communicating (hence inter-dependent) tasks τ, we modeled A as a directed acyclic graph (DAG) G_A, where a node represents a task and a link represents the dependency between two tasks. For any two tasks τ_i and τ_j with a directed link $\tau_i \rightarrow \tau_j$, we called τ_i an immediate *predecessor* of τ_j, and τ_j an immediate *successor* of τ_i. To transform the system A to an equivelant A' with the same functionality and end-to-end timing performance, but independent tasks, A' must have the following properties:

1. the size of the task set are the same before and after transformation, i.e, $|\tau'| = |\tau|$;
2. if $\tau_i \in \tau$, then $\tau_i \in \tau'$ but may have a different rate and deadline;
3. all $\tau_i \in \tau'$ are independent; and
4. if there exists a link l_{ij}: $\tau_i \rightarrow \tau_j$ in A, there is a buffer B_{ij} that is written by τ_i, and is periodically read by τ_j.

In the transformed A', successor tasks are decoupled from their predecessors as they need only access to the buffers. Specifically, the explicit triggering mechanism used to meet the precedence constraints in the original system is replaced by an implicit trigger stored and updated as flags along with the shared buffers. For any dependency $\tau_i \rightarrow \tau_j$, τ_j periodically checks B_{ij} for updates from τ_i. The successor tasks decide when to process their input buffers based on these flags. The correctness, with respect to data-dependency, of the system is therefore preserved, and A' functions the same as the original A.

Figures 3.3 give an example system with dependent tasks. The task graph of the original system contains 4 tasks, where T_3 depends on the outputs of both T_1 and T_2, and T_4 depends on the output from T_3, as shown in Figure 3.3(a). The system is subject to two timing constraints C_{14} and C_{24}. The runtime scenario of the origianl system is shown in Figure 3.3(b).



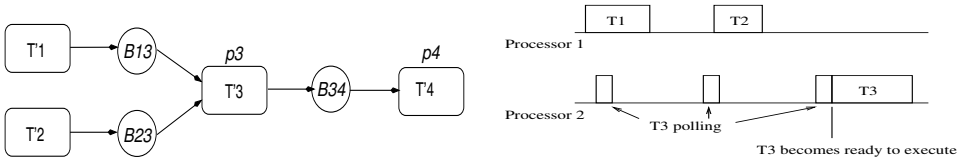
(a) Original task graph with 4 tasks.

(b) Scheduling with dependencies on 2 processors.

Figure 3.3: An example system with dependent tasks.

The system can be transformed into one with independent tasks using shared buffers, as shown in Figure 3.4(a). The transformed system contains the original 4 tasks and 3 shared buffers. The polling

rates p_3 and p_4 are assigned to T_3' and T_4' such that constraints C_{14} and C_{24} are satisfied. Tasks are triggered independently, as shown in Figure 3.4(b).



(a) Task graph with independent tasks after transformation.

(b) Schedule of transformed tasks on 2 processors.

Figure 3.4: The system after transformation using shared buffers.

In addition to preserve the functions, the transformation from A to A' must also be done in such a way that the schedulability and timing constraints of the original system A are preserved in A'. In other words, if A' is schedulable, then so is A. [2] In our approach, this is achieved by deriving the *polling periods* for τ' based on A's timing constraints and schedulability considerations. A *polling rate* is the frequency at which a successor task is invoked to check the update of the shared buffer. As the transformation only changes the internal triggering mechanism of the dependent tasks from event-triggered to time-triggered, only the timing constraints of successor tasks need to be modified. The timing properties of the rest of the system like the input rate of data into the system, or the output rate of data to the environment remains the same.

3.3.1 Derivation of polling rates

The polling rate derivation is a process of reassigning the invocation rates of the successor tasks in a way such that timing and schedulabiity constraints are met after the transformation. The basic idea for deriving the polling rates is to compute the maximum allowable processing time for each successor task. Given an end-to-end timing constraint C_A, the maximum allowable processing time pt_i for a task τ_i is the largest duration τ_i may execute while C_A is met. This includes the delays of τ_i from its buffer updat until the buffered data is processed. If there are n sequential tasks, the sum of pt_i for these tasks should be less than C_A. We can then ensure C_A is met if the maximum delay incurred in executing τ_i is within its maximum allowable processing time. If we assign the maximum allowable processing time as the polling period of the task and makes the A' schedulable, we can ensure every invocation of τ_i completed within the maximum allowable processing time.

The maximum allowable processing time of task τ_i consists of the worst-case update-detection delay of τ_i and the worst-case execution time of τ_i. For any successor task, the worst-case update-detection delay occurs when the shared buffer is updated immediately after the task checks it. Since the task can only detect the updates at its next invocation in such cases, the delay can be at most the duration of one task's period. Therefore, the polling period at which a successor task in A' should be running to satisfy C_A can be expressed as:

$$poll_{\tau_i} + r_{\tau_i} \leq d_{\tau_i} - s_{\tau_i} \tag{3.2}$$

where $poll_{\tau_i}$ is the task's polling period, r_{τ_i} is the response time (also called the *completion time*), d_{τ_i} is the deadline for execution of τ_i, and s_{τ_i} is the start time of τ_i. The term $d_{\tau_i} - s_{\tau_i}$ defines the maximum

[2] Note the condition is only sufficient, meaning that if A' is not schedulable, A may still be schedulable. This implies that our approach introduces additional overhead.

allowable processing time for τ_i. s_{τ_i} can be derived from the deadlines of τ_i's predecessors as:

$$s_{\tau_i} = \max\{d_{\tau_k} : \tau_k \text{ is an immediate predecessor of } \tau_i\}. \tag{3.3}$$

Derivation of polling period of τ_i using Equation 3.2 requires the response time r_{τ_i} of τ_i. The response time r_{τ_i} depends on the allocation of τ_i, which, in turn, may depend on the polling period for each task. This circular dependency between the response time and the polling period for each task can be resolved by iteratively determining solutions for both. We can set the initial polling period of τ_i to the period of the transaction, performing task allocation, and obtaining r_{τ_i}. We can then check Equation 3.2 to determine the $poll_{\tau_i}$. For those tasks that fail the check, their polling periods should be revised systematically followed by task reallocation. This process of polling period assignments and task allocation should be iterated until all tasks in τ satisfy Equation 3.2.

If some task τ_i in τ does not satisfy Equation 3.2, it indicates that the worst-case update-detection delay of τ_i is longer than required and should be reduced. Reducing update-detection delay of τ_i consequently reduces its polling period. To reduce the update-detection delay, we defined an "excess" as the amount of time by which τ_i exceeds its maximum allowable execution time. The polloing period of τ_i can then be modified using Equation 3.4:

$$\begin{aligned} excess_i &= p_{\tau_i} + r_{\tau_i} - d_{\tau_i} + s_{\tau_i} \\ poll'_{\tau_i} &= poll_{\tau_i} - excess_i/n. \end{aligned} \tag{3.4}$$

where $poll'_{\tau_i}$ is the new polling period of τ_i, and $excess_i$ is defined as the amount of time by which τ_i exceeds its maximum allowable execution time. n is a predefined constant to determine the step size of the adjustment. The larger value n is, the smaller each adjustment is. However, since tasks' periods have no effect on the scalability of the allocation and scheduling algorithm for fixed-priority system, different values of n results the same the scalability of the allocation and scheduling algorithm. Thus, we randomly chose $n = 2$ in this work to allow large adjustment at each step.

Algorithm 9 provides the algorithm for polling period derivation. Since shortening the polling periods increases the system workload in each iteration and will eventually lead to an unschedulable task set, the algorithm is guaranteed to terminate in a finite number of iterations.

After polling periods are derived for all successor tasks, schedulability analysis is required to ensure the system-wide timing correctness. Since successor tasks have variable execution times —short execution times without buffer updates and have long execution times with buffer updates, the schedulability analysis algorithm using worst-case execution times can not provide accurate solution. To be able to apply existing schedulability analysis algorithms, we replaced each successor task by two tasks: one dedicated to buffer polling with the execution time for only polling operations, and the other to data processing with the execution time of data processing. The task for data processing is only activated when the shared buffer is updated, and is released with a fixed offset that equals to the polling period to ensure the data availability. These two tasks can then be considered as independent tasks in the schedulability analysis.

3.4 Analysis and verification

After the runtime model is generated, we must verify whether all timing constraints can be met. Although the runtime system is generated with consideration of meeting all timing constraints, these constraints may be violated due to the resource contention, scheduling policies, and overheads of RTOS. To verify that the resources are sufficient for the designed ECSW, and timing constraints are met for all transactions, the analysis must be applied for every resource and every task.

Algorithm 9 Algorithm of iterative polling period derivation.

INPUT: task set τ;
a set of processors P;
end-to-end timing constraints C_A
$poll_{thresh}$: user specified polling period threshold for tasks in τ.
OUTPUT: task set τ with successor tasks assigned a polling period $poll_{\tau_i}$.
BEGIN
distribute timing constraints C_A over each task in τ in A.
foreach $\tau_i \in \tau$ **do**
  $poll_{\tau_i} = p_T$;
end-foreach;
Loop:
  allocate and schedule tasks τ' on processors $P \rightarrow Alloc$;
  if ($\tau_i \in \tau$ can be successfully scheduled) **then**
    if (τ_i satisfies Equation 3.2) **then**
      return $successful$, $\{poll_{\tau_i}\}$;
    else
      foreach τ_j fails Equation 3.2 **do**
        decrease $poll_{\tau_j}$ using Equation 3.4;
        if $poll_{\tau_j} \leq poll_{thresh}$ **then**
          return $failure$;
      end-foreach;
      goto *Loop*;
    end-if-else;
  end-if;
else return $fail$;
END.

In this work, we performed both local analyses and global analyses. The local analyses focus on individual processor and each task. The metrics to check include the CPU usages for application tasks, system overheads, and unused resources. The local analyses also provide timing details of each task, including its response time and times of being preempted. The global analyses focus on end-to-end response times for transactions including executions on involved processors and network communication delays.

To perform these analyses, we need to determine the scheduling policy first. In this work, we assumed all the tasks are scheduled using a static priority-based preemptive scheduler. Such a scheduler is the most commonly scheduler implemented in current commercial and research RTOS such as VxWorks/OSEKWorks, QNX, and RTLinux. Schedules generated using other static scheduling policy, such as the cyclic executive in Time-Triggered Architecture, can be mapped to a schedule using the priority-based preemptive scheduling policy [4].

All analyses are based on time demand analysis of the system. The timing demand analysis is also called busy period analysis, and it is a powerful tool in real-time scheduling theory. We chose timing demand analysis for the analysis and verification because the timng demand analysis is applicable for more general cases with less restrictions compared with traditional schedulability analysis methods like rate-monotonic or deadline-monotonic schedulability analysis.

3.4.1 Local schedulability analysis

The local schedulability analysis computes the response time of every task on every processor, resource utilization for every activity in the system, and available resource left on each processor. The algorithm is a modified version of the algorithm developed by Harbour, Klein, and Lehoczky [10] to include underlying supportive system overheads, such as overheads for timer signal processing and context switches. These overheads can consume significant portion of resources according to our experiments. This algorithm is chosen because it supports the analysis independent of how the priorities of tasks are assigned, whether precedent constraints exist, and the relations between deadlines and periods.

The basic concept in the local analysis algorithm is computing the worst-case phasing for every task. To do so, the components in each task can be viewed as subtasks whose priorities can be assigned internally based on their intermediate deadlines and the task's priority. This means for a given tasks τ_i the priorities of its subtasks in τ_i are non-decreasing in their execution order. Such assigned priorities satisfy the *canonical form* assumptions of the algorithm in [10]. The algorithm is outlined in Algorithm 10.

Algorithm 10 Algorithm for local schedulability analysis.

INPUT: runtime model $M_r =< \tau, l >$;
OUTPUT: schedulability decision (YES/NO);
resource consumptions U, U_i;
response time $resp_i$.
BEGIN
1 **foreach** processor P **do**
2 **foreach** task τ_i on P **do**
3 transform every task into canonical form;
4 determine task set that can preempt or block τ_i;
5
6 compute worst-case phasing from all tasks other than τ_i;
7 determine the number of instances of τ_i in its busy period, N_i;
8 **for** 1 to N_i **do** compute $resp_{\tau_i}$;
9 $resp_i \leftarrow max\{resp^k_{\tau_i} : 1 \leq k \leq N_i\}$;
10 **if** $resp_i > D_i$ **then return** NO: unschedulable;
11 $U_i \leftarrow \frac{wcet_{\tau_i}}{period_{\tau_i}}$;
12 **end-foreach**;
13 $U_p = \sum_{\tau_i} U_i + U_{overhead}$;
14 **end-foreach**;
15 **return** YES: schedulabe, U_p, U_i;
END.

In this algorithm, the examined task is first transformed into a canonical form where the priority of the first subtask is the lowest among all subtasks, denoted as p_i. To determine the task set that can preempt and block an examined task τ_i, we classified all tasks on the same processor P into 5 types:

- *Type 1* task set contains tasks that can preempt τ_i. All subtasks of any task in this set have higher priorities than p_i, and can preempt τ_i multiple times.
- *Type 2* task set contains tasks with some subtask that has a higher priority than p_i, followed by subtasks with lower priority than p_i. A task in this set can preempt τ_i only once.

- *Type 3* task set contains tasks with some subtask with a priority higher than p_i, followed by some subtask with lower priority than p_i, then a subtask with a higher priority than p_i again. A task in this set can introduce both one preemption and one blocking to τ_i.
- *Type 4* task set contains tasks with some subtask with a lower priority than p_i, followed by a subtask with a higher priority than p_i, then a subtask with lower priority than p_i again. Only one task in this set can introduce blocking time to τ_i.
- *Type 5* task set contains tasks with subtasks whose priorities are all lower than p_i. The tasks in this set can be ignored in the analysis, as they have no effect on τ_i execution.

The worst-case phasing happens when τ_i initiates the τ_i-level busy period. The worst-case phasing can therefore be computed using the blocking and preemption times caused by other tasks. According to the classification of tasks, we recorded the blocking time introduced by Type 3 and Type 4 tasks, and chose the maximum one as the blocking time B_i.

To include the overheads of underlying system, specifically timer overhead and scheduling overhead, [3] we modified the original algorithm as follows: given a system running with timer resolution R, the overhead of a timer signal processing δ, and scheduling overhead η for each context switch, an additional task τ_t should be added to the task set with the highest priorty, period $P_t = 1/R$, and execution time $e_t = \delta$. The context switch overhead is included in the preempting task execution time in the computation. The length of τ_i busy period can then be derived using Equation 3.5.

$$\begin{aligned} L_i &= min(t = B_i + \\ &\quad \sum_{\tau_k \in Type1 \cup Type2} \lceil \frac{t}{P_k} \cdot (e_k + 2\eta) \rceil + \\ &\quad \sum_{\tau_k \in Type3 \cup Type4} (e_k + 2\eta) + \\ &\quad \lceil \frac{t}{P_i} \cdot e_i = t) \end{aligned} \tag{3.5}$$

The number of instance of τ_i in $\tau - i$-level busy period can then be computated using Equation 3.6.

$$N_i = \lceil \frac{L_i}{P_i} \rceil \tag{3.6}$$

The response time of the kth instance of τ_i can be iteratively computed from its first subtask until the last subtask using Equation 3.7.

$$\begin{aligned} resp_{ij+1}(k) &= min(resp_{ij}(k) + \\ &\quad \sum_{\tau_p \in Type1 \wedge Type2} (\lceil \frac{t}{P_p} \rceil - \lceil \frac{resp_{ij}}{P_p} \rceil) \cdot (e_p + 2\eta) \rceil + \\ &\quad \sum_{\tau_p \in Type2 \wedge Type3} min(1, \lceil \frac{t}{P_p} \rceil - \lceil \frac{resp_{ij}}{P_p}) \cdot (e_p + 2\eta) + \\ &\quad e_{ij+1} = t) \end{aligned} \tag{3.7}$$

[3] Scheduling overhead in this work include both overhead for scheduling algorithm to decide which task to run and the context switch overhead to actually save the old task's environments and initialize the new task's environments.

Note in Equation 3.7, the j is the indicator of subtasks in τ_i. The task set of $Type1, 2, 3, 4, 5$ change at each iteration. This is because when the subtask j is processed, and $j+1$ is examined, the priority of the substask may increase, and therefore may make some tasks originally in Type 1 (or 2 or 3 or 4) moving to lower priority types that won't preempt the subtask. For the first subtask of τ_i, its response time $resp_{i1}$ can be computed by:

$$\begin{aligned} resp_{i1}(k) &= min(B_i + \\ &\sum_{\tau_p \in Type1 \wedge Type2} \lceil \frac{t}{P_p} \rceil \cdot (e_p + 2\eta) + \\ &\sum_{\tau_p \in Type3 \wedge Type4} (e_p + 2\eta) + \\ &(k-1) \cdot e_i + e_{i1} = t) \end{aligned} \tag{3.8}$$

After obtaining the $resp_{in}(k)$ for the last subtask n of τ_i, we can decide the schedulability by checking the following condition:

$$max((k-1) \cdot P_i + D_i - resp_{in}(k)) > 0 \text{ for all k} \tag{3.9}$$

If Equation 3.9 is met for all tasks on all processors, the system is schedulable, and the algorithm returns a YES answer. The worst-case response time of τ_i is then computed as $resp_i = max_{1 \le k \le N_i}(resp_{in}(k))$. The utilization of each task can then be computated as $U_i = \frac{e_i}{P_i}$, while the total processor utilization of processor P is $U_p = \sum_{\tau_i \in P} U_i$. The resource consumed by the OS service can also be derived by:

$$U_s = R \cdot \delta + \sum_{\tau_j \in H_i} \lfloor \frac{resp_i}{P_j} \rfloor \cdot 2\eta$$

If Equation 3.9 is not satisfied, the algorithm returns NO for unschedulable. By checking the response times and utilizations of individual tasks and services, the designer can pinpoint which activity consume more resources, and can modify design to reduce it accordingly. The computation complexity of computing the response time of τ_i during its busy period is $O(n^2 L/(1-U))$, where n is the number of tasks on the processor P, L is the ratio of the longest period to the shortest period in the task set, and U is the utilization of the task set. If there are m processors, the complexity is $O(mn^2 L/(1-U))$.

The delays of the communication links can be computed using the same algorithm by treating each message transmission as a task/subtask in the system.

3.4.2 Global timing analysis

The global timing ananlysis is required to check the satisfaction of the end-to-end timing constraints. The local schedulability analyses study only the tasks on a given processor, and verifies whether the given deadlines for the task set can be met. However, since the information processing flow in a large scale embedded system like mission computing in weapon systems and automotive vehicle control typically involve consecutive tasks running on different processors through different communication links, satisfying schedulability and timing constraints on each and every processor does not guarantee the end-to-end timing constraints of information processing flows can be met. This is because the interferences among different transactions can cause resource contentions on different processors and communication links, therefore results in violating the end-to-end timing constraints of the chain. To this end, we adopted the end-to-end schedulability analysis algorithm based on [26] for timing analysis of end-to-end task chains after local schedulability and timing constraints are verified.

The method for the global end-to-end timing analysis is still busy period analysis. Instead of computing only busy period of a task on one processor, we must compute the busy period of a transaction involving all tasks across all participating processors, and the interferences introduced by tasks of other transactions sharing these processors. In the global timing analysis, we treat the consecutive tasks on the same processor as one in the analysis. We further assumed that the tasks of a transaction on different processors sychronize their executions through modified-phase-modification procotols (MPM) [26]. MPM is a dynamic task release control protocol. In the MPM protocol, a task instance is triggered by a timer signal if its predecessor completes before a time bound. Otherwise, the task is triggered directly by the completion signal of its predecessor. As the task synchronization across different processors relies on message passing in current RTOSes, MPM can be easily implemented by modifying the message receiver process to control the release time of the synchronized task.

The basic idea of the global end-to-end timing analysis algorithm is to compute the worst-case response time of every transaction, and compare it with the end-to-end deadline. According to the MPM protocol, the release interval between any two consecutive instances of a task τ_i is no less than the task's period P_i. Therefore, we treated all tasks in a transaction as periodic tasks whose periods equal to the transaction's period in the worst-case response time computation. In such a case, the worst-case response time of a transaction T_i is the sum of the worst-case response times of all its constitute tasks on every processor.

An issue of the algorithm in [26] is that the aglorithm can only analyze a transaction with a straight chain of tasks. However, in our model, the task graph generated by Algorithm 8 may contain fork- and join-branches, and tasks on different paths can be running on different processors in parallel. Therefore, instead of simply adding all tasks worst-case response times to obtain the transaction's worst-case response time, we chose the longest execution path of the transaction, and summed the worst-case response times of the tasks on the longest execution path as the transaction's worst-case response time. Note the communications between tasks were also considered in the computation of the transaction's worst-case response time. Since the communications are triggered by the tasks, and we assumed the messages are only sent at the end of a task computation (last operation in the task) and are only received at the beginning of a task (the first operation of the task), we modeled the communications as dependent tasks that are successors of message generation tasks and predecessors of message reception tasks.

The global timing analysis algorithm is outlined in Algorithm 11.

Algorithm 11 first computes the worst-case response time of every task in an examined transaction T_i. Since a task is statically allocated to one processor, the worst-case response time computation is exactly the same as it is in the local schedulability analysis. So the computation can be done using Equation 3.5 for L_p computation, Equation 3.6 for N_p, and Equation 3.7 and 3.8 for $resp_{ip}$. After obtaining the worst-case response times of all tasks in T_i, we used $resp_{ip}$ as a weight for each task in the task graph of T_i, and find a path with the maximum weight as the longest path LP_{T_i}. The worst-case response time $resp_{T_i}$ of T_i is then the sum of tasks' worst-case response times on LP_{T_i}. $resp_{T_i}$ is then compared with the specified end-to-end deadline D_{T_i}. If $resp_{T_i}$ is longer, the global timing constraints is violated. The designer needs to make changes to reduce the end-to-end response time of T_i. If every transaction completes all its tasks before its end-to-end deadline, the algorithm returns a YES decision with the worst-case response time for every transaction, indicating the global timing constraints are met with the current system configuration.

3.5 Automatic design refinement

Automatic design refinement adjusts the timing and scheduling parameters of tasks in the runtime model in order to meet the timing and resource constraints when the system analysis returns with $fail$

Algorithm 11 Algorithm for global timing analysis.

INPUT: runtime model $M_r =< \tau, l >$;
OUTPUT: decision on meeting end-to-end deadlines (YES/NO);
worst-case response time $resp_{T_i}$ of transaction T_i.
BEGIN
foreach transaction T_i in M_r **do**
  foreach $\tau_i \rightarrow \tau_j \in l$ **do**
    map l to τ_l with delays as e_l and period of τ_i as period of τ_l;
    $T_i \leftarrow \tau_l$;
  end-foreach

  foreach $\tau_{ip} \in T_i : 1 \leq p \leq m$ **do**
    compute worst-case phasing from all tasks on the same processor P;
    compute the worst-case τ_{ip}-level busy period L_p;
      determine the number of insteances N_p of τ_{ip} in its busy period L_p;
      for 1 to N_p compute $resp_{ip}$;
      $resp_{ip} \leftarrow max\{resp_{ip}^k : 1 \leq k \leq N_p\}$;
      assign weight of $\tau_{ip} : w_{ip} \leftarrow resp_{ip}$;
  end-foreach

  $LP_{T_i} \leftarrow find_longest_path(T_i, w_{ip})$;
  $resp_{T_i} \leftarrow \sum_{\tau_p \in LP_{T_i}} resp_{ip}$;
  if $resp_{T_i} > D_{T_i}$ **then return** NO, $fail$;
end-foreach

return YES, $resp_{T_i}$;
END.

results. As discussed in previous section, when both local schedulability analysis and global timing analysis return YES, the system is schedulable, and all timing constraints are satisfied with the given system setup — processors contain components and tasks, priorities and execution rates assigned to components and tasks, and their release offsets and deadlines. The software can then be coded accordingly for the target for implementation. However, if the analysis returns with NO results, some modification must be made before the software can be implemented. In general, the change of any one or more system design parameters of the designed ECSW, such as replacing components with ones requiring less resource (shorter execution time or shorter message), reorganizing the software architecture with more parallelism and/or fewer communications and dependencies, relaxing the timing constraints, adding more resources (increase number and/or capacities of processors and links), and choosing different scheduling policies and priority assignments, may lead to satisfaction of timing and schedulability constraints. Although many of these changes requires human intervention, we can focus on only a small number of factors to alter, and use a guided-search to iteratively refine the results at each step in hope to find a feasible solution.

In this work, we considered only approach of refining tasks' rates and priorities automatically in order to generate a schedule that satisfies all timing and resource constraints. Other approaches such as altering components with the same functionality but different performance, modifying software architecture,

and changing the platform should be done by the designer manually.

Period transformation. Period transformation is a method used to alter the invocation rate of a task. Such a transformation can change the resourse utilization of the system and response times of both the task and the transaction that contains it. The effects of the transformation are reversal for resource utilization and response time. Increasing the invocation rate makes a task execute more frequently, therefore it may result in shorter response, but higher resource utilization. On the contrary, decreasing the invocation rate can extend the task's response time, but lower the resource consumption.

In this work, we tranformed tasks' periods only when the system resources provided in the platform are not sufficient. Insufficient resources are reflected in a fail result from local schedulability analysis with some processor utilization greater than the specified bound. To reduce the resource usage, we chose lengthening the task period if the local schedulability analysis fails. This indicates the insufficient resource to schedule task set on a processor, thus requires to lower the resource utilization. This approach works only under the assumption that the task's rate can vary within a range. The control quality may be better if a higher rate is used, but is acceptable at the lowest rate bound. The acceptable rate range is usually specified as a Quality-of-Service parameter of the task.

There are two issues in the period transformation: (i) which task's period to lengthen, and (ii) by how much. In general, we should transform the period of the task that causes the system to be unschedulable. Since the unschedulability is a combinational results of all tasks sharing the resources, it is difficult to determine which one is the root reason. If the designer defines the importance of the performance impact for each task, we can lengthen the period of the task causing the least performance degradation. Hoever, it is usually difficult for a designer to determine such importance for each task. In the case that the designer could not provide the information, we used the tasks' utilizations to assist the selection. We chose the one with the highest utilization to transform first. To determine the new period, we adjusted the period linearly to minimize the performance impact of the task, as in Equation 3.10.

$$\begin{aligned} P^{i+1} &= P^i + d \\ d &= \lfloor (P_u - P_c)/n \rfloor \end{aligned} \tag{3.10}$$

In Equation 3.10, we allowed the period transformed only between the task period upper bound P_u and current period P_c. This is because the transformation is only performed when the task set is unschedulable, and only periods longer than the current period may make the set schedulable. The floor operations in d computation is to ensure obtained period is an integer. n defines the number of steps to extend the period to its upper bound. Depending on different n, the transformation can lead to a schedulable system that is faster or slower. In our implementation, n was chosen to be 5. If $d = 0$ under $n = 5$, d was assigned to 1. The algorithm is shown in Algorithm 3.5.

In Algorithm 3.5, if a task τ_i is unschedulable, only those tasks whose priorities are higher or equal to τ_i, denoted as τ_H, may be extended. This is because the lower priority tasks do not affect the execution of τ_i. Algorithm 3.5 increases the period of the highest utilization task in τ_H one step at a time. After each period adjustment, the task with adjusted period is put back to the task list as a new different task in the furture adjustment. A different task may be selected for transformation in consecutive loops due to the utilization of the task is lowered after the period adjustment. The hypothesis of this transformation strategy is that minimum and even period extensions of multiple tasks result in less control degradation than a large period extension of one task. If, for any unschedulable task, all tasks in τ_H are extended to their period upper bound, but still can not make the task set schedulable, the algorithm returns a $fail$, indicating there is not sufficient resources to maintain the lowest control quality. Human designer's involvment is a must to address this.

Algorithm 12 Period transformation algorithm.

INPUT: task set τ;
acceptable period range (P_l, P_u);
OUTPUT: task set with new period τ;
BEGIN
1 sort τ in descending priorities;
2 **foreach** τ_u unschedulable in order in τ **do**
3 $\tau_H \leftarrow \{tau_i : priority_i \geq priority_u\}$;
4 **while** $(unschedulable) \wedge (\tau_H \neq \emptyset)$ **do**
5 remove τ_i with $P_c = P_u$ from τ_H;
6 $\tau_c \leftarrow \tau_i : max_{\tau_i \ \ \tau_H} U_i$;
7 extend P_{τ_c} according to Equation 3.10;
8 analyze schedulability of τ;
9 **end-while**;
10 **if** *unschedulable* **then return** $fail$;
11 **end-foreach**
12 **return** τ; **END.**

Priority adjustment. Another method to automatically refine the system design in order to make an unschedulable system schedulable is adjusting tasks' priorities. Period transformation can only be used when the lower and upper bounds of tasks' periods are specified. As mentioned before, it is difficult for the designer to accurately determine such bounds for all tasks while still meeting the control requirements. In such a case, the priorities of tasks are only parameters we can alter automatically and effectively. The tasks' priorities in a system are usually assigned according to some strategy such as functional importance and timing attributes (e.g., rate-monotonic assignment, deadline monotonic assignment). In a system using a preemptive fixed-priority scheduling policy, a task τ_i experiences interferences from tasks with priorities higher than τ_i. The long response time of τ_i may be caused by more tasks running at higher priorities than τ_i. Since different tasks have different timing constraints, it is possible to raise the priority of τ_i to shorten its response time, and consequently meet its timing constraints, while still maintaining the satisfaction of timing constraints of other tasks. Our priority adjustment algorithm was developed for this purpose.

The automatic design refinement with priority adjustment is applicable only to the cases that all processors' utilizations are less than their bounds, but the end-to-end response time of some task is not met. In such a case, the platform provides sufficient resources, and the violation of timing constraints may be caused by inproperly assigned priorities.

The method of priority adjustment uses simulated annealing technique [17, 16]. Simulated annealing is a heuristic search with a predefined energy function. At each step, only a solution that reduces the value of the energy function is examined. Heuristic search of the solution is essential in this domain because (i) a large number of tasks exist in the runtime model, therefore we need a scalable solution; and (ii) there are information dependencies among tasks within a transaction and across transactions. This implies that adjusting priorities of current task assignments so that the task set become schedulable is as difficult as finding a priority assignment to make the task set schedulable from scratch, which is NP-hard.

For a given task set with some timing constraints violated, the priority adjustment chooses a proper task and adjusts its priority. Similarly, as in the performance transformation, there are key issues: (i)

which task we should choose, and (ii) what the new priority should be. There are two strategies on choosing which task to adjust. Given a transaction T whose end-to-end deadline is not met, we can raise the priority of some task in T, or lower the priorities of some high priority task in transactions other than T, so that the task of T can run with less interference from other transactions on each processor. In this work, we chose to raise the priority of a task in a transaction with a missed deadline. We only chose the lowering strategy when all tasks of the transaction with missed deadlines run at the highest priority on every processor. Among the tasks in an examined transaction T, we adjusted the priority of a task τ_i that satisfies:

1. τ_i is on the critical path of T;
2. the successor of τ_i has a higher priority;
3. τ_i has the longest response time among all tasks satisfy the previous 2 conditions.

To determine a new priority of τ_i, we first convert the task set τ to a canonical form [9] for each transaction. The conversion ensures that the end-to-end execution order of tasks in a transaction is maintained if the task set is still in canonical form after the priority adjustment. Since the priorities of tasks in canonical form are non-decreasing along the task execution order, if we find a task τ_i whose priority p_i is less than its successors p_{i+1}, and adjust its priority no higher than p_{i+1}, we can maintain the task set in canonical form, and shorten the response time of τ_i as a result of its priority increase. This may consequently shorten the end-to-end resposense time of the transaction. In this work, we chose raising the task's priority by 1 at each time to minimize the impact to other transactions. Such increment stops when the task priority equals to its sucessor's.

To evaluate whether the new assignement yields a better response time, an energy function is introduced. Considering both timing constraints and system overheads, our energy function is defined as:

$$E = k_1 * max\{resp(T_i) - d(T_i)\} + k_2 * n_{p-level} \tag{3.11}$$

where E is the energy value of the new assignment; T_i is the examined transaction; $resp(T_i)$ and $d(T_i)$ are the response time and timing constraint of T_i, respectively; $n_{p-level}$ is the number of distinct priority levels. k_1 and k_2 are constants. They should be chosen in a way such that the first term and second term are weighted the same in E.

The energy function in Equation 3.11 was evaluated for every priority adjustment, including the initial assignment. If an adjustment results in a lower value of E, and all timing constraints are met under the new priority assignment, the process stops. Otherwise, the adjustment continues with the new assignment. On the other hand, if the E results in a higher value, we marked the task as priority unchangeable, and selected another one.[4]

The termination of the process is determined by a feasible solution is found and neither energy increases or decreases have been made for the last N_E adjustments. In our implementation, we set $N_E = 3$.

After each priority adjustment, we need to recheck the timing constraints and schedulability of the system.

[4]In simulated annealing, it is possible that a non-solution point have less energy than a solution point. Theoretically, we should allow a jump regardless to the energy increases. However, such a jump should be governed by a function which is always decreasing so that the backward jumps are less frequent and finally approach zero after the initial stage. Since we didn't have such a govern function, we simply disallowed backward move.

Chapter 4

Techniques for RTOS Measurements

As real-time operating systems (RTOSs) are maturing, most ESW tend to use the existing RTOSs for runtime resource management. Similar to component modeling for application software, we modeled an RTOS as an integratoin of a set of functional components, called *services*, each of which provides a service to the application. To support the analysis of the final software, the performance characteristics of RTOS services must be known. Although RTOS vendors provide some product performance characteristics such as throughput, interrupt latencies, and context switch time, such information is usually insufficient for ESW timing and schedulability analysis. System designers may need to measure the RTOS performance to meet their domain modeling and analysis requirements. Such measurements are expected not only to reveal the interested information of services, but also to be done systematically so that the measured results can be reused for various analyses of a family of applications. Since most RTOSs are released only as binaries, the measurements are expected to be done without source code. Furthermore, the measurements should be made on per service basis since "services" are the basic OS unit used in ESW integration. A traditional measurement method meeting these requirements is benchmarking. However, benchmarks generally yield results dedicated to only a single, statically configured workload on the system. Thus, multiple benchmarks must be run for all interested application configuration, resulting in a costly duplication of measurement effort.

In this work, we developed an end-to-end measurement method based on a combination of microbenchmarks and synthetic workloads. The end-to-end (e2e) measurement is a run-time, sampling-based method, which records the start time and the end time of an activity of interest as an external observer. Such a measurement methodology is known to introduce the minimum intrusion into the measured system [18], and provide the performance results close to what the application will experience at runtime. The e2e method can also uncover inter-service dependencies and performance metrics without RTOS source code, and can be applied to any system-level service like middleware and some subsystem services. Microbenchmarks [5, 6] are an effective method for measuring individual and independent OS-level services without instrumenting the kernel. However, as typically used, they exercise the system in a limited way that is not necessarily representative of an actual application workload. We derived more realistic performance metrics by coupling the microbenchmarks with representative, domain-specific synthetic workloads. Synthetic workloads allow the measurements to be taken under conditions close to the real applications with representative resource usage and interaction patterns. Through such synthetic workloads, our measurements can cover most frequently-used application configurations and interaction patterns so that the results can be reused to analyze a family of applications.

This developed technique of measurements has been applied to the performance measurements of two fundamental RTOS services, namely timing services and scheduling services. The performance

characteristics of a service are defined as a set performance metrics for the measurements. The measurements are made on selected RTOSs and under varying workloads used for both OEPs. Specifically, the platform we measured included QNX on Intel Pentium processor (experiment platform for both Boeing OEP and Automotive Vehicle-to-Vehicle control), RTLinux on Intel Pentium processor (candidate platform for Beoing OEP), and OSEKWorks on MPC 555 (experiment platform for Automotive Powertrain Control).

Before design the experiments to measure the RTOS service metrics, we must understand how the measurement results can be used. In this work, we assume the performance model of the designed ESW can be constructed hierarchically, reflecting realization layers [2], as shown in Figure 4.1. The performance of a component/subsystem at a higher layer depends on the interactions of the components at the same and immediately adjecant layers, and the performance of components immediately below it. This is because the components at the same layers and layers immediately above provide the workloads for the system, while the layer immediately below provides the services. In this paper, we treat all software components running on top of an OS as applications, although a finer-grained model with more layers representing other system software, such as middleware, can be constructed in the same way. With this modeling method, after the ESW is constructed hierarchically by integrating components, and the performance characteristics of the constitute components are annotated to the componnents, the performance model can be constructed immediately for analysis.

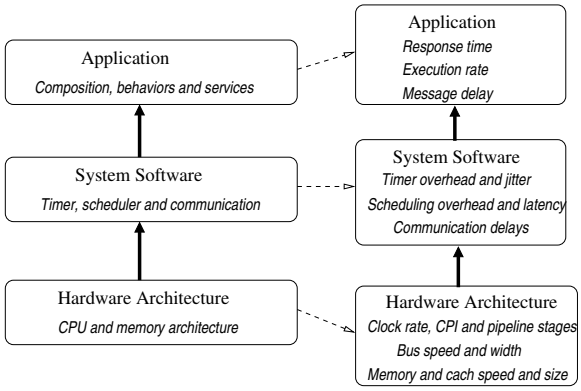


Figure 4.1: Hierarchical, analytical performance model.

This layered model facilitates our OS-level performance measurements and modeling because the performance dependencies among components are broken down into layers. The measurements of OS service performance can now be done for each service of interest individually with its possible usage patterns in ESW. Performance model construction and analysis of embedded software will then rely only on the measured performance of OS services and application functional design. In this paper, the performance of OS-level services are measured directly, although it can also be derived from the performance of hardware components and low-level OS services using, for example, lmbench [19, 6].

As with microbenchmarks, a set of experiments will be generated to measure each service. Since the performance dependencies among components at different levels are broken down in our hierarchical/layered performance model, the measurements of OS service performance can now be done individually for each service of interest. The experiments of service performance measurements include "representative" service requests and application configurations. The measured values will be collected at the application level. Since the existence of design patterns in a given domain usually leads to a small number of likely application configurations, synthetic workloads can be used to generate such represen-

tative application configurations with good coverage. To consider the effects of hardware architecture, our measurement will be applied directly to each combination of hardware and RTOS. Although the measured results of service performance will be hardware-specific, techniques [19, 6] exist to break the dependencies further.

In the following sections, we detailed our experimental design and measured results of timing service and scheduling service on the selected platform. The discussion is intended to illustrate how the method works and how to apply it. The applications of the measurement results in performance modeling and analysis are discussed later in Chapter 6 with evaluations.

4.1 Design of Experiments to Measure Timing and Scheduling Services

We treated the timing service and scheduling service as a component in the system software layer in Figure 4.1 to design experiments for measurements. To minimize the interferences from other activities in the operating system, unnecessary services such as networking were turned off during the measurements. Timing services include various clock and timer management mechanisms implemented in a RTOS. The performance information for such a service is critically important to time-based activities in ESW. Scheduling services, on the other hand, are essential for software execution. The performance of such a service plays a key role in assessing the quality of the entire system. Timing and scheduling services are the basic services required by all embedded control applications and are supported in all RTOSs.

Since timing and scheduling services normally run as part of the RTOS kernel at the highest priority in the system, applications can experience large overheads and unpredictability due to these services. Such overheads and unpredictability may cause problems on meeting the application-level timing constraints and achieving stable control [33, 31]. Our measurements of timing and scheduling services should, therefore, reveal how these overheads and unpredictability change with different system configurations and application usages. When performance analysis needs to be done, such measured information can then be used to construct an accurate performance model based on the given application usage and system configuration.

To achieve such a measurement goal, we defined a set of performance metrics for overhead and unpredictability which were reusable for the analysis of a family of applications. We then constructed synthetic workloads to enumerate representative ways of using timing and scheduling services in ESW applications and measured the service performance using microbenchmarks. The measurements were done on the selected hardware and RTOSs. Note that the performance of different hardware and RTOSs were not measured to compare and select targets. Instead, they were measured for constructing performance analysis models for a family of applications that would be executed on these targets.

4.1.1 Measurement strategy

Measurement environment. The performance of RTOS services is hardware-dependent, as the hardware provides the execution environments for the measured RTOS. Here we did not assume the existence of methods for deriving RTOS service performance from the hardware performance and configuration. Instead, we designed a set of experiments for each different combination of hardware and operating system to obtain the performance of RTOS services directly. Since different RTOSs have different implementations and provide different ways of using these services, it is important to learn the effects of such differences on performance. This was one of the considerations during our decision on the platform selection. Table 4.1 lists the hardware we used in the measurements.

The RTOSs we measured include QNX 4.24, OSEKWorks 2.0 and RTLinux 3.0. We assume that the source code of these kernels is not available for the measurement, although the source code availability of RTLinux 3.0 helps us understand the relationships among the OS services and can be used to verify our analysis results. The collected data was temporarily stored in the main memory and was later

Hardware	Processor type	Processor spped (MHZ)	Memory size (MB)	Cache size (KB)	Bus speed (MHZ)
Intel P133	Pentium	133	32	128	66
Intel P166	Pentium	166	32	128	66
ETAS	MPC 555	40	2	-	20

Table 4.1: Hardware configurations for OS service measurements.

dumped to an appropriate host for analysis after the services used for storage were turned on at the end of each measurement.

The versions of selected RTOSs that we had could not be run on all the hardware platforms. The configurations used for measurements are given in Table 4.2.

Hardware	QNX	RTLinux	OSEKWorks
P133	QNX-P133	RTL-P133	-
P166	QNX-P166	RTL-P166	-
MPC555	-	-	OSEK-MPC

Table 4.2: Testbed configuration for measurements.

Performance metrics. It is important to choose the performance metrics for a given OS service carefully. The metrics are expected to be minimum in number, reusable for a family of applications, and independently measurable. A minimum set of metrics is desired to reduce the cost of experimentation and data collection while still providing sufficient data for the analysis. Finding a minimum set of performance metrics for a service requires understanding of the analysis requirements, dependencies among the OS services, and the relationships between different metrics. Reusability means that the metrics are measured once and reused whenever the same environment is used, thus eliminating duplicate measurements for the same environment for different applications with similar workloads and interaction patterns. Finally, the independently-measurable metrics simplify the experiments and data analysis and are more flexible when they are used in a performance model.

For the timing service, we measured clock overhead and interval jitter. The clock overhead is the CPU time used to process each signal generated from the system clock. The interval jitter is the variance in the length of time intervals. Interval jitter affects when periodic operations actually occur and is thus a source of unpredictability in the system. For the scheduling sevice, we measured the context switch overhead. The context switch overhead is defined loosely as in [6], which is the time taken from terminating a task to starting execution of another ready task.

Measurement tools and analysis. Almost all current measurement methods [13, 18] are based on monitoring events in the system under evaluation. Measurement tools using events can be classified as event-driven tools, tracing tools, sampling tools, or indirect measurement tools [18]. Event-driven tools record event occurrences. Tracing tools are similar to event-driven tools but record more system information that can be used to uniquely identify each event occurrence. Sampling tools record the system state at a fixed time interval instead of recording every event asynchronously upon its occurrence. Indirect measurement tools are designed in an *ad hoc* manner, and used only when the metrics cannot be directly measured. Among these tools, event-driven and tracing tools require instrumentation of the

OS services, thus making the measurements intrusive. Although such tools can provide information on detailed activities in the measured service, their intrusiveness makes the measurement results inaccurate, while the details of activities inside the measured service are usually unnecessary for application performance analysis. Indirect measurements are usually not used whenever direct measurements are possible.

We use sampling tools to measure the specified performance metrics. The advantages of sampling tools include: (a) no kernel instrumentation is required, (b) the measured performance is close to what an application will experience at runtime, and (c) the least amount of intrusion is introduced. The primary disadvantage is that they can only provide statistical results. Other tools, such as logging events and generating traces, also exist, but they require instrumenting the service software and can introduce perturbations in the performance. Furthermore, the sampling tool gathers information based on external observations, which is suitable for use in application performance analyses since applications are also external observers for RTOS services.

To obtain accurate timing values, our measurement tool sampled the processor clock cycles for each measurement. Most modern processors are equipped with some registers dedicated to performance and timing measurements. We used the hardware Time-Stamp Register (TSR) on the Pentium processor [12] and the Time-Base Register (TBR) on the MPC 555 [20]. Both are 64-bit registers, initialized to 0 when the system powers up and incremented by 1 upon every hardware clock tick at the CPU speed.

A statistical analysis method was used to process the measured data. For each experiment, 10,000 samples were collected during the normal execution. In addition to computing the average and standard deviation of the measured parameters, we also found the maximum and minimum values as performance bounds for each measured parameter on a given platform.

4.1.2 Experiment design

To correctly measure the performance of the selected services and make them reusable, we considered application workloads in experiment design. The workload at the application level can make the measured services behave with different levels of performance. Measurements with workloads representing various application usages can then make the measured results reusable. Since each of our measurements corresponded to one combination of hardware and RTOS, the experiment design focused only on generating application-level workloads. The workloads were designed to cover a set of representative application usages after investigation many typical applications in machine control, avoinics mission computing, and automotive engine control software.

Experiments for timing service measurement: the metrics of timing services included clock overhead and interval jitter. The clock overhead depends mainly on the clock resolution. It can be measured by executing a test program under different resolutions. Specifically, given a program P with execution time e, the overhead of each clock tick can be computed using Eq. (4.1).

$$e_m = e + I_0 \cdot o + I_1 \cdot o + I_2 \cdot o + \ldots \tag{4.1}$$

where e_m is the measured execution time of P; e is the real execution time of P; and o is the overhead of processing each clock tick. Each term $I_i \cdot o$ represents the overhead of processing the clock ticks during the time interval $I_{i-1} \cdot o$. I_i is the coefficient of the i-th order overhead. Given the clock resolution r when the measurement is taken, I_i can be calculated recursively as:

$$I_0 = \lfloor \frac{e}{r} \rfloor, \quad I_1 = \lfloor \frac{I_0 \cdot o}{r} \rfloor, \quad I_2 = \lfloor \frac{I_1 \cdot o}{r} \rfloor, \quad \ldots \quad I_n = \lfloor \frac{I_{n-1} \cdot o}{r} \rfloor, \ldots \tag{4.2}$$

It can be seen from Equation (4.2), I_i decreases exponentially. Thus, given any e, there exists a positive integer n such that for any $N > n$, $I_N = 0$, as the overhead introduced by its previous term will eventually be less then r. In the OS service measurements, the order of I_i seldom exceeds 2 as e is normally tens of milliseconds and the OS overheads are in the order of microseconds. Therefore, the clock overhead can be computed as follows:

$$e_m = e + \lfloor \frac{e}{r} \rfloor \cdot o + \frac{\lfloor \frac{e}{r} \rfloor}{r} \cdot o^2 \tag{4.3}$$

$$e_m = e + \lfloor \frac{e}{r} \rfloor \cdot o, \tag{4.4}$$

Equation (4.3) can be used when the clock resolution is fine (normally less than $0.5ms$) and/or the execution time is long, while Equation (4.4) can be used for the other cases. According to Equation (4.3) and (4.4), the clock overhead can be finally derived using following equation:

$$o = \begin{cases} \frac{\sqrt{4 \cdot r \cdot \lfloor e/r \rfloor \cdot (e_m - e) + r^2 \cdot \lfloor e/r \rfloor} - r \cdot \lfloor e/r \rfloor}{2 \cdot \lfloor e/r \rfloor} & \text{for fine resolution} \\ \frac{e_m - e}{\lfloor e/r \rfloor} & \text{otherwise} \end{cases} \tag{4.5}$$

The execution time e of P is necessary to compute clock overhead o. The measured execution time e_m is usually larger than e since the clock signal will be generated regardless of whether it is used or not, and its overhead is included in the measured e_m. However, since e_m only includes the clock overhead incurred during the e_m measurements, we set the clock resolution far larger than e to obtain a measurement e_0 that is close to the real e, as given in Eq. (4.6).

$$e_0 = e, \qquad for \;\; r \gg e \tag{4.6}$$

In our experiment, the test program P was designed with an execution time of $10ms$. The resolutions used for the clock overhead measurements ranged from $100\mu s$ to $100ms$.[1] The selected clock resolutions and methods to set them are given in Table 4.3.

RTOS	method to set resolution	values (μs)
QNX	clock_setres()	100, 200, 400, 500, 600, 800, 1000, 10000, 100000
RTLinux	rtl_setschedmode()	100, 200, 400, 500, 600, 800, 1000, 10000, 100000
OSEKWorks	Rtcinit()	100, 200, 400, 500, 600, 800, 1000, 10000, 100000

Table 4.3: Resolution setting methods and values for the measured OS.

Interval jitter is a product of both the clock resolution and the application configuration. Clock resolution affects jitter when the RTOS allows the intervals of activities and/or events to be to be not integral multiples of the clock resolution. In such a case, the time of the end of the interval is rounded up to the next clock tick. Interval jitter is affected by the application configuration in a number of ways:

[1] The resolution range was chosen based on the capacity test of a platform and the usage in applications.

the number of independent timers, the interval patterns of these timers, and the priorities of tasks using these timers. The timer resolutions used in our experiments were selected to be 500 μs and 1 ms.[2] The workloads in our experiment were designed to produce different patterns of timer intervals and different numbers of timers. Table 4.4 lists the values used in our jitter measurements.

Factor	values
clock resolution	500 μs, 1 ms
timer patterns	harmonic, non-harmonic
task priority	highest, medium, lowest
number of timers	1, 2, 5, 10, 15, 20

Table 4.4: Factors and values for interval jitter measurements.

A set of test programs was designed to perform the jitter measurements with the listed system attributes. Since only one timer can be associated with a process/thread in all the RTOSs studied, we needed up to 20 tasks in the experiments. Table 4.5 lists the attributes of every experiment task,[3] and Table 4.6 shows the combinations of the tasks of measuring the performance with a different number of timers.

task id	period (harmonic) (ms)	period (non-harmonic) (ms)	priority	task id	period (harmonic) (ms)	period (non-harmonic) (ms)	priority
1	1	2	1	11	80	83	11
2	5	5	2	12	90	89	12
3	10	9	3	13	100	103	13
4	15	17	4	14	150	145	14
5	20	19	5	15	200	211	15
6	30	31	6	16	250	239	16
7	40	43	7	17	300	217	17
8	50	49	8	18	400	395	18
9	60	61	9	19	500	513	19
10	70	71	10	20	600	613	20

Table 4.5: Attributes of experiment tasks.

Experiments for scheduling services measurements: The metric for measuring scheduling services was context switch overhead. We define the context switch time loosely as in [6], i.e., the kernel time spent between when the current running task is terminated and when a new ready task gets to run. The overhead includes the time to terminate the current running task and store its execution environment, the scheduling time to select a new ready task to run, and the activation time to restore the execution environment of the new tasks and give the control to it. These times can not be individually measurable without kernel instrumentation. However, the total elapsed time taken from the termination of one task to the start of a new task is measurable without kernel instrumentation and is

[2]These values were chosen by considering the fact that the clock overhead introduced by these values should be small but potentially has a significant impact on jitter.

[3]In this table, a smaller number was used for a higher priority.

test case	1	2	3	4	5	6
# of timers	1	2	5	10	15	20
task in set	{1} or {10} or {20}	{1, 10} or {1, 20}	{1, 5, 10, 15, 20}	{1, 3, 5, 7, 10, 11, 13, 15, 17, 20}	{1, 3, 4, 5, 7, 9, 10, 11, 13, 14, 15, 17, 18, 19, 20}	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}

Table 4.6: Task combinations for test cases.

closer to what applications will experience at runtime. Therefore, such measurements are more suitable and reusable for application performance analyses.

The context switch overhead depends on the scheduling algorithm, the number of tasks in the ready queue, and the organization of the ready queue (e.g., sorted or unsorted queue). The priority-based preemptive scheduling algorithm is the one supported by all current RTOSs and used most frequently in ESW. So, our context switch overhead measurements were based on this scheduling algorithm. The task set ranged from 2 to 20 tasks. The measurements were taken between two specially-designed tasks in the task set. All other tasks, called *interference tasks*, were introduced only to change the length of the ready queue to learn the effect of the queue length on the context switch overhead. The task set was checked manually to be schedulable before the measurements. Table 4.7 shows these tasks and their attributes used for measuring the scheduling service performance.

task id	priority	period (ms)
1	2	1
2	1	triggered by task 1
3-20	3	1

Table 4.7: Attributes of experiment tasks.

The measurement with the given task set was designed as follows: Task 1 ran periodically with 1 ms period, and triggered Task 2 upon its completion. The priority of Task 1 was lower than that of Task 2, but higher than all interference tasks. The TSR and TBR values were logged at both the end of Task 1 and the beginning of Task 2. Interference tasks ran with the same period as Task 1. Thus, every 1 ms, all tasks but Task 2 were ready and were moved to the ready queue. Since Task 1 had the highest priority in the ready queue, it executed first. After its completion, Task 2 was triggered and was in the ready queue. Similarly, Task 2 became the highest priority task and executed before any other task. Thus, Tasks 3–20 only affected the ready queue length during the measurement, and did not contribute any overhead to the context switch time between Task 1 and Task 2. The context switch overhead could then be obtained from the difference between the pairs of sampled values of Task 1's completion and Task 2's start. All interference tasks were assigned to the same priority since their priorities had no effect on the measurements. Table 4.8 shows the number of tasks used for each measurement to learn the effect of the ready queue length.

test case	1	2	3	4	5
# of tasks	2	5	10	15	20
task in set	{1, 2}	{1, 2, 3-5}	{1, 2, 3-10}	{1, 2, 3-15}	{1, 2, 3-20}

Table 4.8: Task combinations for test cases.

4.2 Measurement Results

4.2.1 Results of clock overhead measurements

The measured execution times of the test program with different clock resolutions are presented in Table 4.9. Note that the original measured execution times were represented as the number of clock cycles, and we converted them to real wall clock times in Table 4.9.

platform	statistics	resolution (ms)							
		0.1	0.2	0.4	0.5	0.8	1	10	100
QNX-P133	average	10.725	10.355	10.171	10.133	10.078	10.065	10.006	10.0005
	maximum	13.871	11.845	10.875	10.674	10.261	10.154	10.015	10.012
	minimum	10.702	10.339	10.157	10.119	10.069	10.057	10.005	10.0002
	std	0.085	0.043	0.018	0.014	0.008	0.006	0.002	0.0002
RTL-P133	average	12.732	11.287	10.676	10.548	10.353	10.301	10.099	10.078
	maximum	20.912	13.589	11.613	11.222	10.754	10.638	10.132	10.092
	minimum	12.714	11.298	10.671	10.544	10.359	10.298	10.098	10.078
	std	0.249	0.012	0.061	0.025	0.015	0.013	0.0012	0.0005
QNX-P166	average	10.876	10.397	10.188	10.144	10.082	10.063	10.006	10.004
	maximum	12.738	11.207	10.483	10.348	10.188	10.121	10.011	10.009
	minimum	10.755	10.362	10.177	10.139	10.077	10.061	10.005	10.0002
	std	0.063	0.055	0.008	0.008	0.004	0.004	0.004	0.0002
RTL-P166	average	12.695	11.223	10.544	10.408	10.251	10.197	10.032	10.018
	maximum	18.523	13.685	11.779	11.373	11.766	10.588	10.061	10.046
	minimum	11.901	10.889	10.443	10.337	10.212	10.169	10.030	10.015
	std	0.249	0.012	0.061	0.025	0.015	0.013	0.0012	0.0005
OSEK-MPC	average	23.755	14.156	11.782	11.398	10.871	10.705	10.149	10.097
	maximum	23.804	14.166	11.812	11.411	10.895	10.723	10.150	10.129
	minimum	23.689	14.108	11.754	11.353	10.836	10.664	10.146	10.090
	std	0.0203	0.0217	0.0285	0.0236	0.0282	0.0261	0.0069	0.0173

Table 4.9: Measured execution times with different clock resolutions.

The clock overhead at each resolution was computed using Equation (4.5). In the clock overhead calculation, we used the minimum execution time when the resolution is set to 100 ms as e_0 for each case. The computed clock overhead for all test cases are shown in Figures 4.2, 4.3 and 4.4.

The measurement results showed that the clock overhead tends to decrease in general as the duration between clock ticks increases. This indicates that a fine-resolution clock will consume more system resources and may cause a schedulability problem, although such a clock may make the system more responsive. The quantitative effects of clock resolutions are OS-dependent. For QNX, the normal overhead is around $5 \sim 7\mu s$. But the maximum can be around $30\mu s$. RTLinux overhead is around $20\mu s$ with the maximum at around $60\mu s$. The overhead for OSEKWorks is around $60\mu s$ except a higher

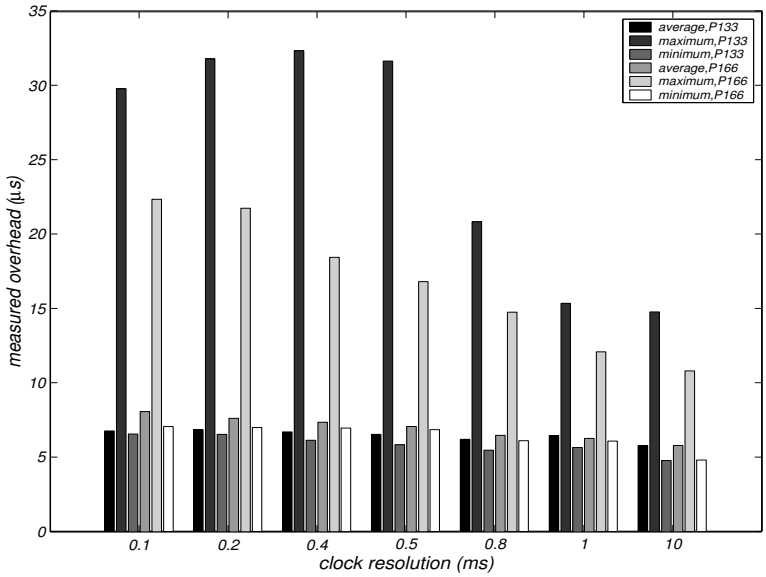


Figure 4.2: Timing service overhead of QNX.

overhead of $70\mu s$ is experienced when the clock resolution is set to 0.1 ms. We also experienced a system hang when the clock resolution was set to smaller than $50\mu s$ for QNX-P133, QNX-P166, $70\mu s$ for RTLinux-P133 and RTLinux-P166, and $80\mu s$ for OSEKWorks-MPC. The measured overhead for QNX and RTLinux is much less than the values required to make the system halt, while for OSEKWorks is very close. This indicates that the minimum available clock resolution depends on both OS implementation and hardware configuration. This may be caused by simple kernel function and hardware configuration of OSEKWorks-MPC platform, but some interrupt and timing related activities in the kernel.

The measurement results also showed that a faster processor can generally reduce the clock overhead. As can be seen in Figure 4.2, using a faster processor reduces the maximum overhead, although it does not improve the average and minimum cases. As for the results of RTLinux shown in Figure 4.3, both average and minimum overheads were reduced on the P166 platform. The small difference between the processor speeds of P133 and P166 along with other hardware factors (bus speed and cache size) may be the reason why similar average and minimum values were measured for QNX and a small improvement was seen for RTLinux. Due to the limited differences between P133 and P166 platform, the hardware effects on the performance need further investigation.

Comparing the overheads of QNX and RTLinux on both platforms with those of OSEKWorks-MPC, the clock overhead of OSEKWorks was almost constant for any given resolution, while the maximum overhead for both QNX and RTLinux was significantly larger than average for any given resolution. The less variant overhead for OSEKWorks may be due to the simple functionality of OSEKWorks [30] and the flat memory structure of the MPC555 [20]. Both help reduce unpredictability during execution.

4.2.2 Results of interval jitter measurements

We then measured the effects of clock resolution, the number and pattern of timer intervals, and task priorities on interval jitter. First, we were interested in how different clock resolutions affect the jitter of different interval lengths. The measured results of interval jitter for QNX and RTLinux under different clock resolutions were plotted in Figures 4.5 and 4.6. For OSEKWorks, we did not observe any jitter for the examined clock resolutions.

According to the measured results, the interval jitter varied greatly from one OS to another. In

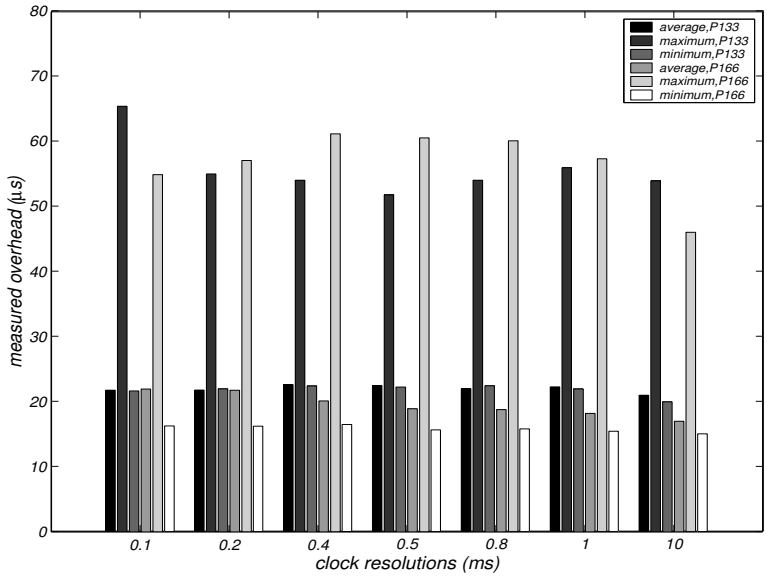


Figure 4.3: Timing service overhead of Real-Time Linux.

QNX, the interval jitter increased as the clock resolution became more coarse, while the jitter for RTLinux showed almost no change with different resolutions. The reason for this could be that QNX uses a clock-based scheduler while RTLinux and OSEKWorks use event-based schedulers. The lack of observed jitter for all experiments with OSEKWorks may also be the result of simple OS implementation and predictable hardware architecture.

The results also showed that the interval jitter is independent of the interval length. This is different from the conventional understanding that the interval should be some relative large multiples of the clock resolution to overcome the jitter. The clock resolution had a distinct effect on the magnitude of the interval jitter. As can be seen in Figure 4.5, the jitter was always bounded by twice the clock resolution. Figures 4.5 and 4.6 also indicate that using a faster processor did not reduce the interval jitter for an OS using a clock-based scheduler, but reduced the jitter for an OS using an event-based scheduler.

Next, we studied the effects of the number of timers and interval patterns on jitter. The measurements also included the jitter experienced by tasks with different priorities. Figures 4.7 and 4.8 plot the measurement results where the intervals are harmonic and non-harmonic on QNX, respectively. Figures 4.9 and 4.10 show the results of the same experiments on RTLinux, while Figures 4.11 and 4.12 show the results of OSEKWorks.

From these results, we first observed that the interval jitter increases with the number of timers in the system for all measured cases. Such dependencies should be an OS property and independent of hardware. Both cases of the same OS running on different hardware and different OSs running on the same hardware showed the same tendency of interval jitter changes. These results suggest that reducing the number of timers by combining tasks with the same intervals would reduce the interval jitter and consequently improve the system performance.

The interval jitter experienced by tasks with different priorities were also significantly different. A higher priority task experienced a smaller jitter, while a lower priority task experienced a larger jitter in all our measurements. Such observations are independent of the number of timers and interval patterns. Larger jitter experienced by a lower-priority task are likely the result of the cumulative effects of kernel

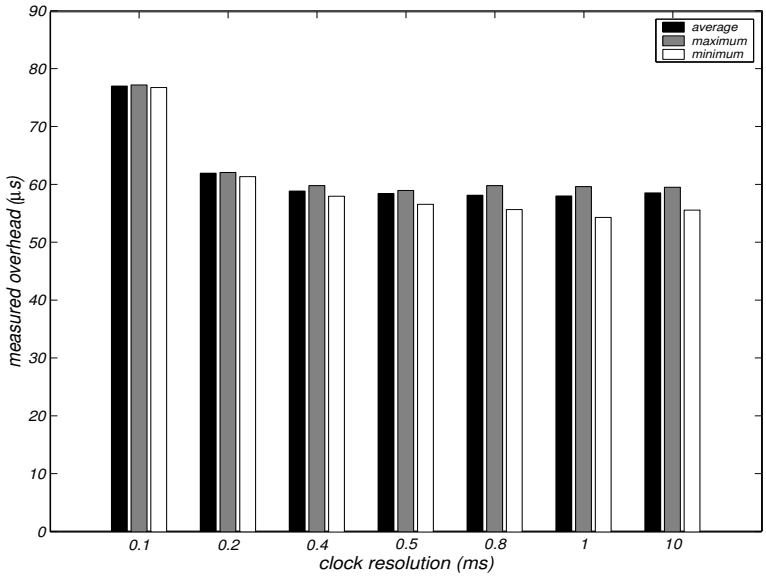


Figure 4.4: Timing service overhead of OSEKWorks.

activities that have a lesser effect on higher-priority tasks.

The interval patterns also had significant impact on interval jitter, and the impact depended on the of OS structure [15] and the number of timers in the system. Among the measured cases, jitter was almost the same for both harmonic and non-harmonic intervals when there were a relatively small number of timers (≤ 5). When the number of timers became larger, the jitter with non-harmonic intervals became larger for QNX, but showed the opposite for RTLinux and OSEKWorks. This implies that the clock-based OS implementation favors harmonic intervals, while the event-based OS implementation favors non-harmonic intervals.

The "memory effects" of timer intervals found in our previous research [33] were also observed in these measurements. However, the effects showed up differently: some longer intervals were followed by a sequence of shorter intervals to compensate it or vice versa in our measurements, instead of a longer interval followed immediately by a shorter interval to fully compensate it.

We also observed that a larger jitter was present for experiments on the P166 board with a faster processor for both QNX and RTLinux. This is counter-intuitive in that a faster processor should yield better performance. This could be either the hardware itself (e.g., different manufactures of some chips and boards) or the implementation of RTOS (e.g., different rates of some OS internal activities on different processors). This requires further investigation.

4.2.3 Results of context switch measurement

The context switches were measured under the different configurations as described in Section 4.1. To learn the relationship between clock resolution and context switch time, we took measurements under different clock resolutions; specifically, 0.5 ms and 1 ms, as the OS scheduler can be either clock-based or event-based.

The measured context switch times of QNX, RTLinux, and OSEKWorks are shown in Figures 4.13, 4.14, and 4.15, respectively. For QNX, the average and minimum context switch times were not sensitive to the number of tasks in the ready queue, but the maximum context switch times increased when the number of tasks in the ready queue increased under a finer clock resolution. The difference between the system with 20 interference tasks and the system without any interference tasks can be as high as

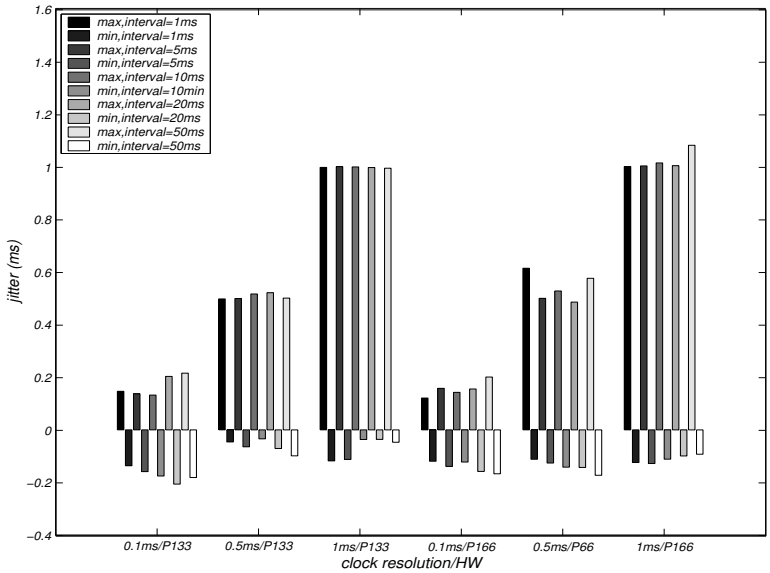


Figure 4.5: Interval jitter of QNX.

300%. For RTLinux, both average and maximum context switch time increased as the number of tasks in the ready queue increased under any clock resolution, while the minimum times remain the same. Both average and maximum context switch times of the system with 20 interference tasks was twice as high as those for the system without any interference tasks. The context switch time for OSEKWorks showed little difference. These results imply that the context switch time depends heavily on the the RTOS implementation, and at least, will not increase if the number of tasks is reduced.

We also observed that the clock resolution had a significant impact on context switch time. The context switch time (average, maximum and minimum) with a small clock resolution was higher than that with a larger resolution for QNX, while it was the opposite for RTLinux and OSEKWorks. This is because the context switch time of a clock-based scheduler may be more sensitive to the clock overhead, while the event-based scheduler may be more sensitive to the resolution.

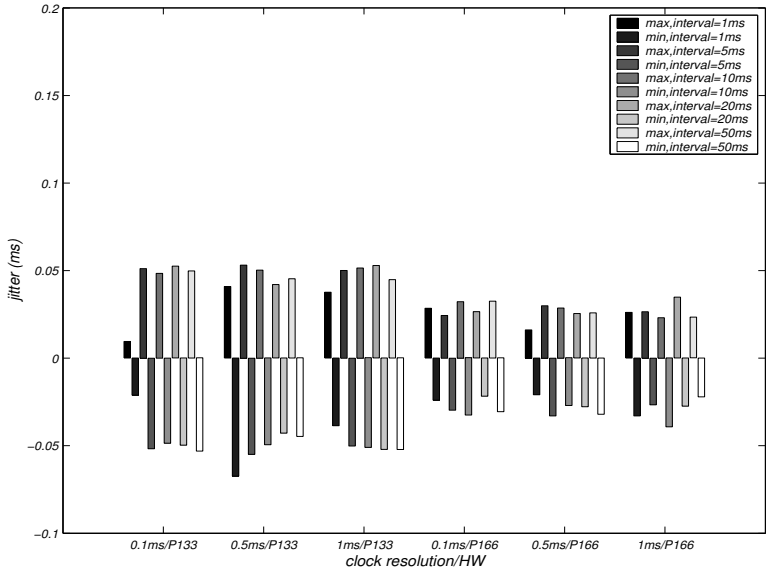


Figure 4.6: Interval jitter of RTLinux.

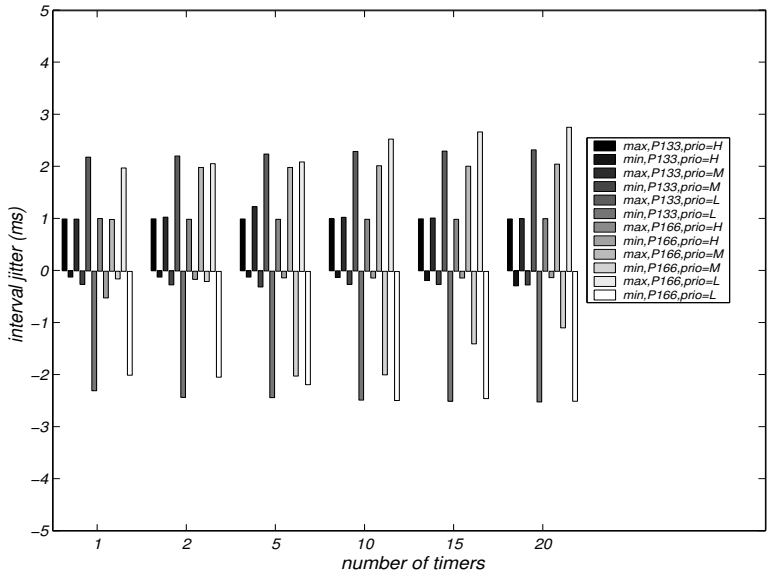


Figure 4.7: Interval jitter for harmonic intervals on QNX.

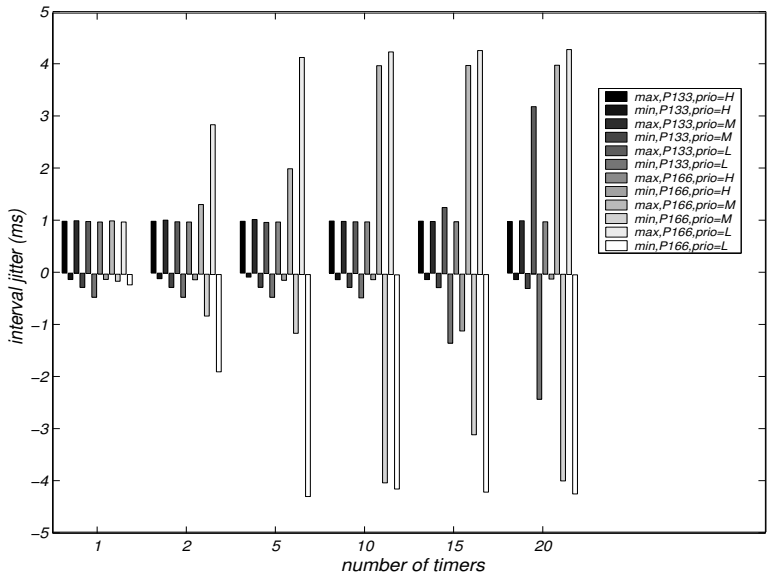


Figure 4.8: Interval jitter for non-harmonic intervals on QNX.

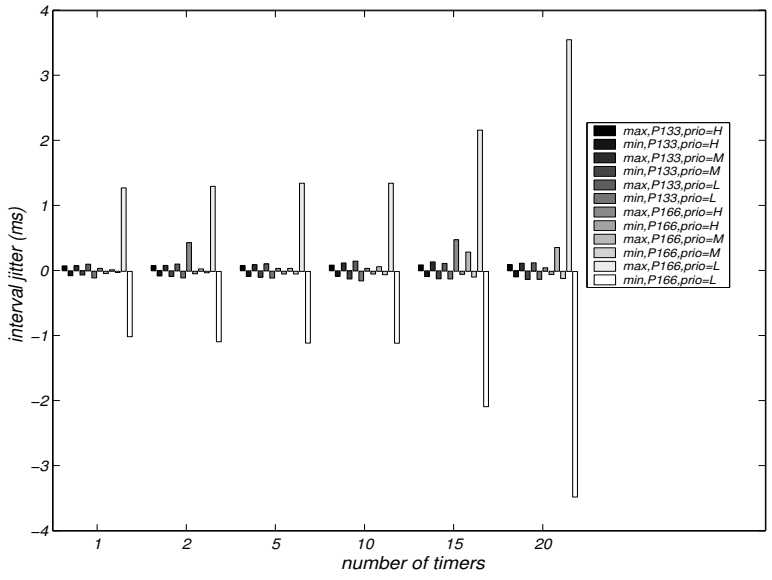


Figure 4.9: Interval jitter for harmonic intervals on RTLinux.

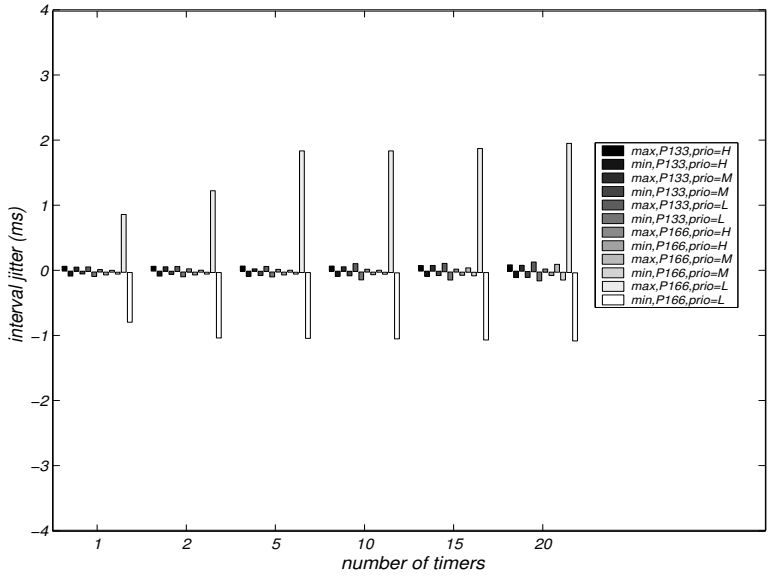


Figure 4.10: Interval jitter for non-harmonic intervals on RTLinux.

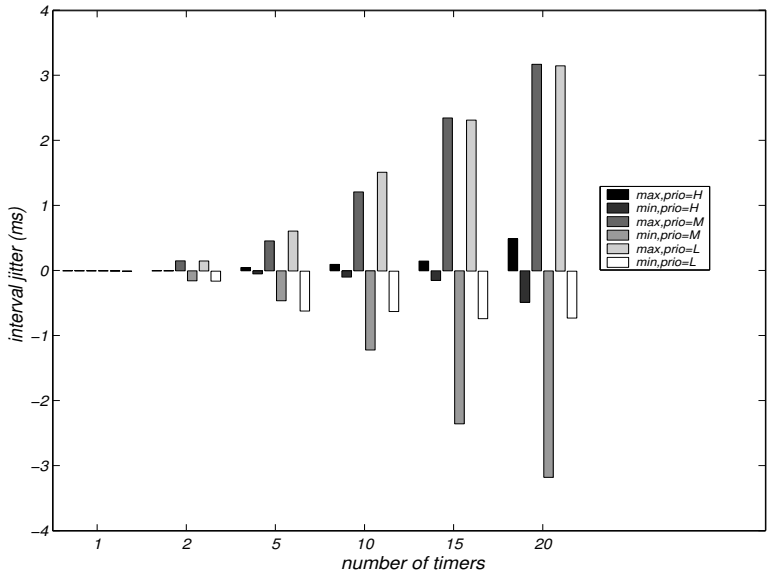


Figure 4.11: Interval jitter for harmonic intervals on OSEKWorks.

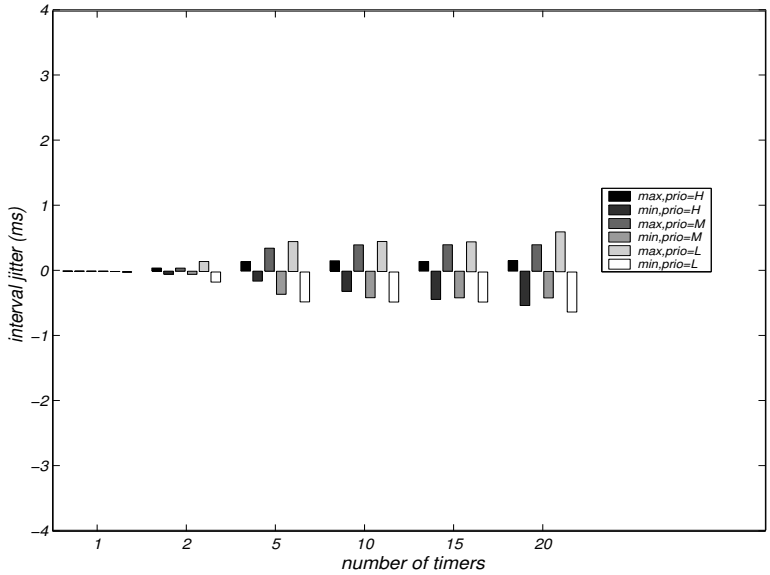


Figure 4.12: Interval jitter for non-harmonic intervals on OSEKWorks.

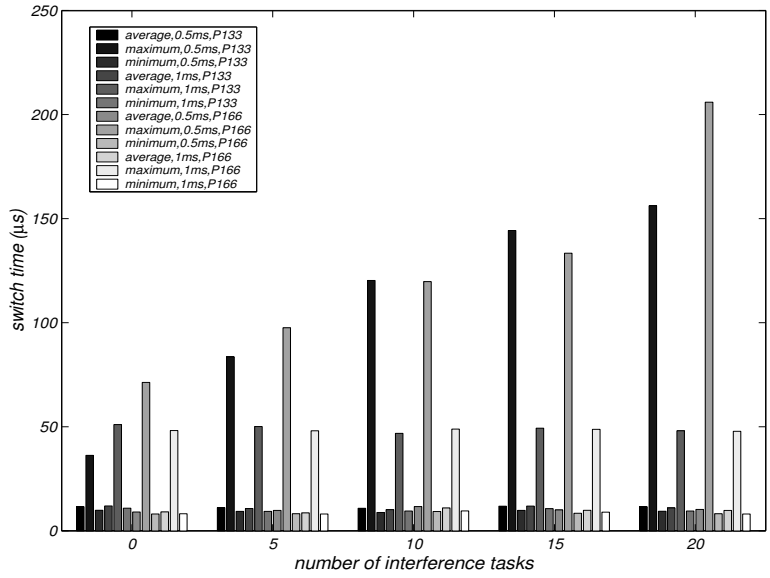


Figure 4.13: Measured context switch time for QNX.

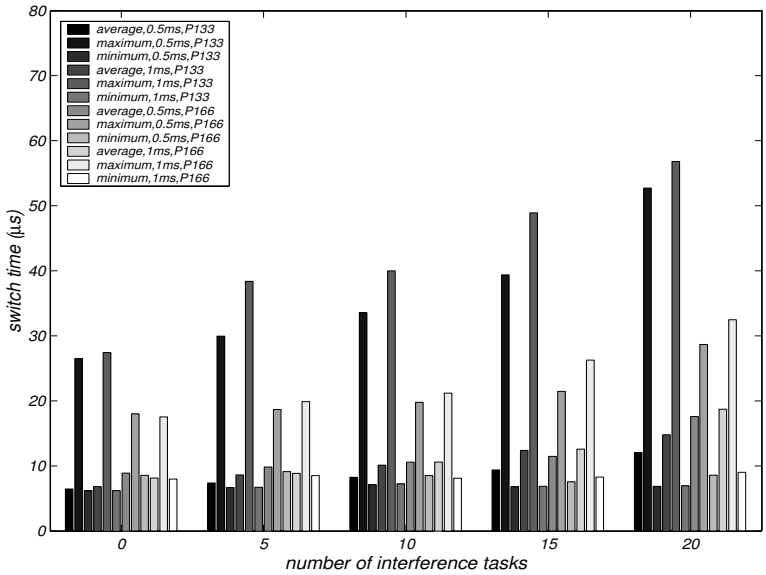


Figure 4.14: Measured context switch time for RTLinux.

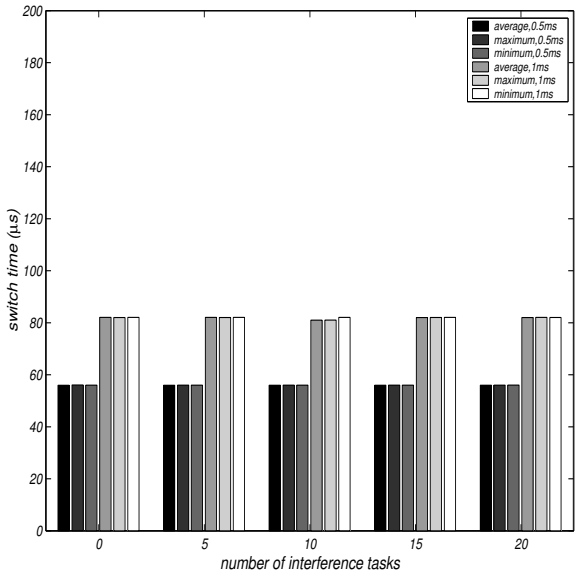


Figure 4.15: Measured context switch time for OSEKWorks.

Chapter 5

AIRES Tool Implementation

To facilitate the use of developed method and algorithms, we integrated all techniques described in previous sections to form the AIRES toolkit to help ESW designers analyze timing and schedulability and provide design assistance at multiple design phases along the tool chain. The AIRES tooklkit was implemented in the Generic Modeling Environment (GME) [11], GME is a MS Windows-based software tool and provides a graphic modeling interface for system modeling and analysis. Integrating with GME allows the AIRES toolkit to interact with designers graphically.

The components of the AIRES toolkit include meta-models, analysis algorithms based on the meta-models, and a built-in development process. The meta-model was constructed using GME built-in meta-modeling mechanism, including model elements (called atoms in GME), their properties, visulability, and constraints. These modeling elements are used as building blocks for creation of application models. Analysis-related modeling information, such as transaction invocation rates, end-to-end deadlines, execution times of application components, and OS overheads, were specified for these models when the designers created them. The implementation of our analysis algorithms needs to parse the model, extract both structure and attributes of the components and models, and provide the analysis results.

All developed algorithms were implemented as either interpreters in GME or standalone programs taking the model files in a common exchange format. An interpreter, also called model interpretreion in GME, is one of the mechanisms to access GME models and generate useful information if necessary. There are two approaches to implement an interpreter: through COM interferces or through the Builder Object Network (BON). The COM interface provides the means to access models, attributes and connectivities through a GUI. The BON interface maps all modeling and modeled structures as a builder object. A BON interface is implemented in C++, it provides users the flexibility of any implementations in C++ programming language. We implemented our algorithms and developed methods using the BON interface due to the fact that our algorithms and methods do not focus on GUI operations but generating new information out of the models. The shortcome associated with using the BON interface is that all the implementations are then meta-model-specific. Therfore, a new implementation is necessary for a new meta-model. However, since the meta-model is relatively stable in a domain, the meta-model-specific implementatin is acceptable.

Our algorithms were developed to form a tool chain with the built-in development process. In other words, the output of one algorithm can be fed to another algorithm for the successive design phase as its input. The built-in development process enforces designers to follow certain design steps to detect design errors and refine the models at each steps during ECSW development. Following such a built-in process ensures that the essential information for analysis are completed at each design phase, and some

design details can be generated automatically with the consideration of multiple constraints. There will be fewer design errors during the development, and thus the development time can be reduced. The built-in development process is shown in Figure 5.1

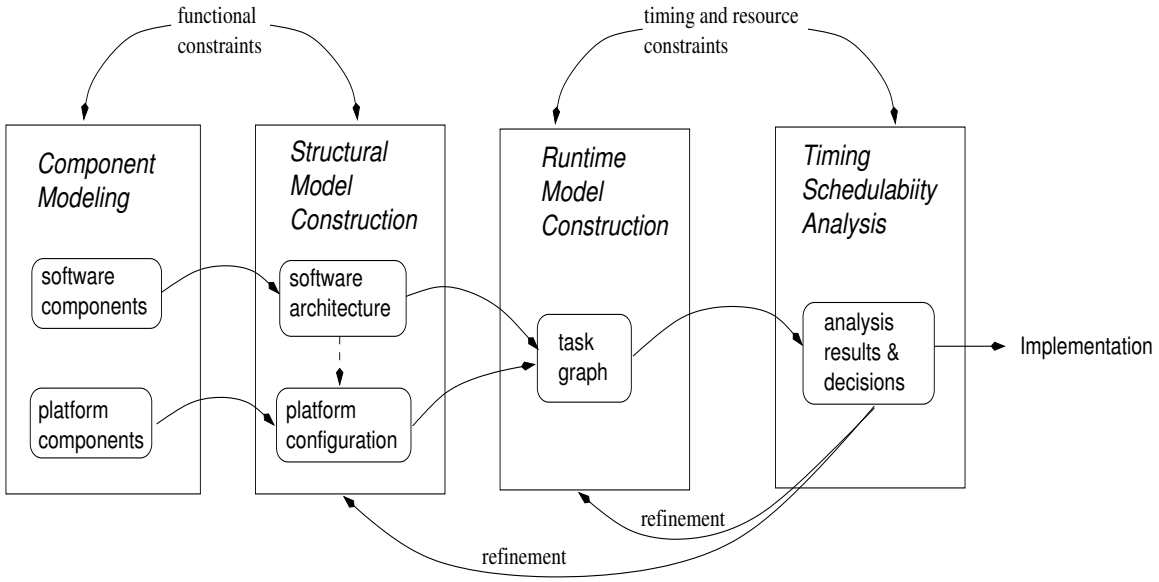


Figure 5.1: The design process built-in AIRES toolkit.

In the built-in design flow, the components' models must be constructed first. Component's models included both functional software components such as various data processing functions, device drivers, and control algorithms, and platform components such as processor boards, I/O boards, networks, and RTOSs. The performance characteristics of components should be measured and stored with their models. Platform components can be profiled and measured beforehand using the techniques described in Chapter 4. The software components can be measured during their unit tests. Construction of components' models does not necessarily start from scratch. Any software and platform constructed for existing systems can be reused, as far as the required specifications are completed in the existing models. The AIRES toolkit can be used to construct the components and build component libraries for reuses in various application design. The AIRES tool further completes the unspecified component properties with default value assignments.

Given an application, we need to first determine the software architecture that satisfies the application's functional requriremnets. Meanwhile, the platform must be designed to provide proper execution environments for the designed software. Both are achieved by selecting components and building the interconnections among them to achieve the function objectives of the application. The platform component selection and configurations are determined not only based on the components' functionality, but also with consideration of the software workloads. A tradeoff between the platform capacity and cost must be made at this phase. In other words, it is always desired to provide a platform with sufficient capacity for the given software workloads but with minimum cost. Some algorithms such as first-fit allocation and communication-minimization can be used to help to decide how many resources are essential. In them AIRES tool, the software architecture and platform configuration are done through dragging-and-dropping components from corresponding model libraries to a workspace. The functional analysis and verification such as signal composibility and cyclic dependencies can be performed for the software architecture model. The software architecture model should also contain transactions of the

system derived from the system behavioral models, which describes the execution scenarios. As performance characteristics are annotated to the components, the component selection and integration process should also be a performance-aware process, i.e., the components and their connections should be made in a way that consumes the smallest amount of the resources as possible. Such software architecture will satisfy the performance constraints better while reduce the cost of the platform.

The runtime model is generated after the software structural model and platform have been generated. All algorithms on component allocation, timing attributes' assignments, and task formation algorithms are applicable to this phase. The result is a task graph with information of execution allocation, invocation frequencies, and inter-task communications. The timing and schedulability constraints come into the picture at this phase to help on decisions of scheduling to meet these constraints. In the AIRES toolkit, generation of the runtime model from the software and platform models can be automated using the previously discussed algorithms.

Timing and schedulability analysis follows the runtime model generation. This step takes a task graph generated in the runtime model generation phase, and analyzes the satisfiability of all timing and resource constraints, both locally and globally. If all constraints are met, the generated task sets with their runtime attributes can be delivered for implementation. Otherwise, modifications of the models and/or system attributes must be made. The obtained analysis results should be used to direct the modification. In the AIRES tool such modifications, called *refinement*, can be done automatically if period transformation and/or priority adjustment are allowed.

It is common in current ESW development that multiple tools based on different model assumptions from different vendors are used in the design and analysis tool chain. To extend the integratability and applicability of AIRES toolkit, the AIRES algorithms were implemented in a way such that they can be used as needed with other tools. According to the nature of these algorithms, we partitioned the algorithms into 3 categories: composite check, runtime system generation, and schedulability analysis. Each category contains a group of related algorithms and was packaged as either one separate interpreter or a separate menu option in the standalone program. To integrate AIRES with other modeling and analysis tools in the development tool chain, we have two options:

1. Re-implement the algorithms in the target modeling and analysis tool environment. Given a target tool environment, the algorithms in AIRES tool can be rewritten to take the models in the target tool, parse them to extrace needed information, generate the results, and present the results in the target environment.

2. translate the models/results generated by other tools to AIRES. The algorithms in AIRES were all implemented based on a common data structure that defines necessary information to run the algorithm. Given the models generated by other modeling tool, or analysis results from an upstream tool, we can translate the information required (and only the information required) in the data structures, and import them to the AIRES tool. Similarly, the output from the AIRES tool can also be translated into some format required by other downstrem tools.

In the current AIRES tool implementation, we took the first approach to implement the tool for Automotive application, and took the second for Avionics applications. Although the same algorithms and built-in design flow were used, this choice was made to show the adaptiveness of the proposed approaches as well as facilitate collaborations with our partners.

5.1 Tool implementation for Automotive Applications

The AIRES toolkit for Automotive applications included a meta-model AIRES and three interpreters. The meta-model was constructed in GME. As discussed in Chapter 2, the models were organized in the

following folders:

- *Simulink*. This folder contains the models converted from Simulink diagrams in .mdl files. The components in this folder have been translated into our port-based component structure. The components are used as building blocks, and can be instantiated in application model construction. The connections in the original Simulink diagram were rebuilt when the model was imported into GME.

- *Stateflow*. Similar to the *Simulink* folder, this folder stores the models converted from Stateflow diagram in .mdl files. The components in this folder were not used in current AIRES implementation, but will be in the future to support identification of concurrent activities and transactions. The components in this folder are also reusable in an application model construction.

- *HWFolder*. A model in this folder specifies the platform configuration, including hardware, network, and operation systems. The analysis algorithms, such as component allocation, task formation, and timing/schedulability ananlysis, refer to this model to obtain the execution environment to proceed.

- *SWFolder*. A model in this folder defines the structural model (software architecture) of a designed application software. Models in this folder can be constructed by creating instances or references of those in Simulink/Stateflow folders.

- *TaskFolder*. A model in this folder defines the runtime model of the system. Models in this folder can be either manually created or automatically generated using the algorithms implemented as interpreter. The timing/schedulabiity analysis are implemented using models in this folder.

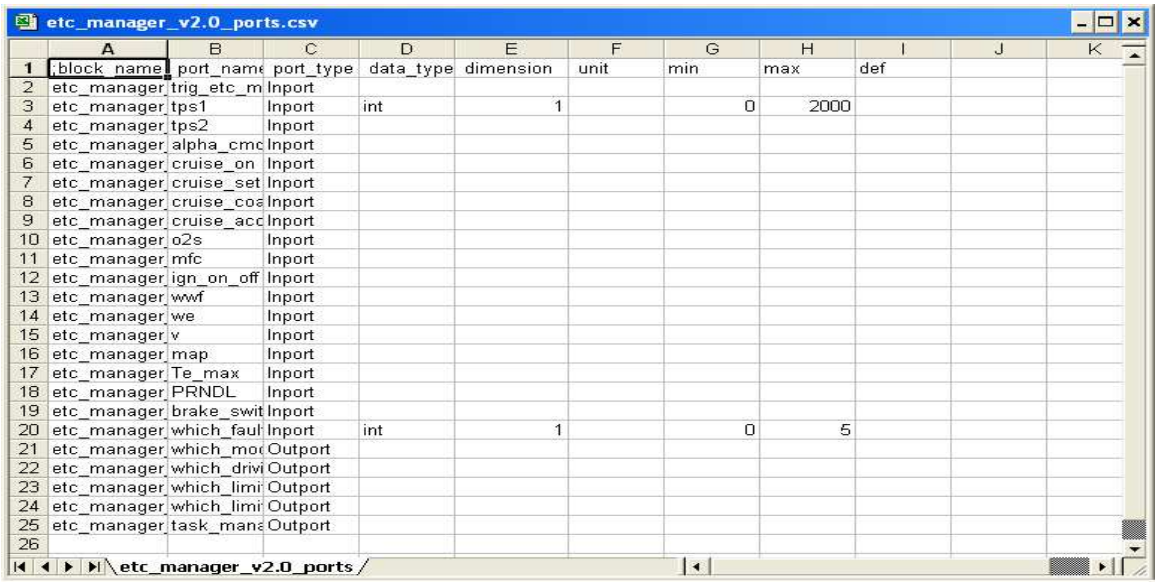
We implemented three interpreters for model transformation and analysis. They are composite.dll for model importation and composition check, comp2task.dll for runtime model generation, and schedule.dll for timing and schedulability analysis.

Composition check interpreter. The composition check contains two functions: translating the Matlab Simulink /Stateflow model into the AIRES model in GME, and checking signal compatibility of a constructed application model. The model translation was implemented based on the UDM parser from Vanderbilt University. Given Simulink/Stateflow diagrams created with the Matlab Simulink tool and stored in a .mdl file, invoking the *Simulink import* function converts the models into AIRES models in GME and creates the *Simulink* and *Stateflow* folders if they do not exist. During the model conversion, the blocks are translated into components, and the signal ports are translated into input/output ports of the components. All block hierarchies are maintained in the translated model. The names for signals, block, and ports are also kept.

The signal composition check compares the linked components' ports to detect any inconsistent linkage. The inconsistent linkages include mismatches of signal types, data types and sizes, and value ranges of variables. Such checks require the attributes of components and ports to be specified beforehand. In the AIRES tool, such specifications are provided as a spreadsheet with the following structure:

<block_name, port_name, port_type, data_type, parameter_dimension, value_unit, value_range, default_value>

Figure 5.2 and 5.3 show two examples of port specifications in this form for an Electronic Throttle Control (ETC).

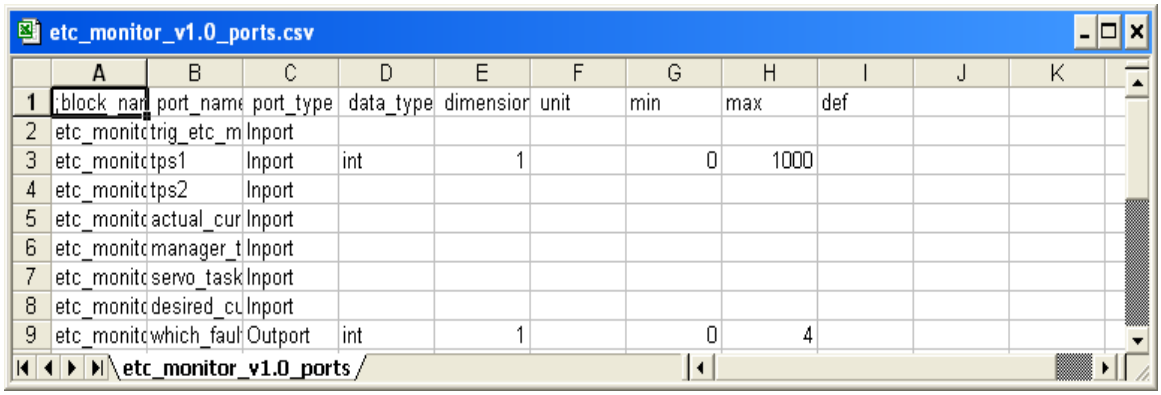


etc_manager_v2.0_ports.csv

	A	B	C	D	E	F	G	H	I	J	K
1	;block_name	port_name	port_type	data_type	dimension	unit	min	max	def		
2	etc_manager	trig_etc_m	Inport								
3	etc_manager	tps1	Inport	int	1		0	2000			
4	etc_manager	tps2	Inport								
5	etc_manager	alpha_cmc	Inport								
6	etc_manager	cruise_on	Inport								
7	etc_manager	cruise_set	Inport								
8	etc_manager	cruise_coa	Inport								
9	etc_manager	cruise_acc	Inport								
10	etc_manager	o2s	Inport								
11	etc_manager	mfc	Inport								
12	etc_manager	ign_on_off	Inport								
13	etc_manager	wwf	Inport								
14	etc_manager	we	Inport								
15	etc_manager	v	Inport								
16	etc_manager	map	Inport								
17	etc_manager	Te_max	Inport								
18	etc_manager	PRNDL	Inport								
19	etc_manager	brake_swit	Inport								
20	etc_manager	which_faul	Inport	int	1		0	5			
21	etc_manager	which_mo	Outport								
22	etc_manager	which_drivi	Outport								
23	etc_manager	which_limi	Outport								
24	etc_manager	which_limi	Outport								
25	etc_manager	task_mana	Outport								
26											

etc_manager_v2.0_ports

Figure 5.2: Port specifications for ETC manager block.



etc_monitor_v1.0_ports.csv

	A	B	C	D	E	F	G	H	I	J	K
1	;block_nar	port_name	port_type	data_type	dimensior	unit	min	max	def		
2	etc_monitc	trig_etc_m	Inport								
3	etc_monitc	tps1	Inport	int	1		0	1000			
4	etc_monitc	tps2	Inport								
5	etc_monitc	actual_cur	Inport								
6	etc_monitc	manager_t	Inport								
7	etc_monitc	servo_task	Inport								
8	etc_monitc	desired_cu	Inport								
9	etc_monitc	which_faul	Outport	int	1		0	4			

etc_monitor_v1.0_ports

Figure 5.3: Port specifications for ETC monitor block.

The structured spreadsheet containing the port attributes must be loaded into the system before checking the signal composition. This is achieved by a *Signal import* button in the interpreter. Multiple invocations are required if the port specifications are stored in multiple spreadsheet files. The signal composition check can be invoked by a *Signal check* button in the interpreter. The check runs Algorithm 1 to check the matches between linked ports. For example, two incompatible signals, *tps1* and *which_fault*, were detected in the ETC model.

Runtime model generation interpreter. The runtime model generation algorithms were implemented in the *Comp2Task.dll* interpreter. The interpreter works with models stored in *SWFolder* and *HWFolder*. Both models have to be constructed manually by the user. To make the AIRES tool easy to integrate with other tools in the tool chain, we allowed the component allocations to be specified partially and manually. The manually specified components allocation are kept unchanged in the later analysis, and only those that can be freely allocated are manipulated by the component allocation al-

gorithm. Note that in both cases, the algorithm uses the processors and network links defined in the *HWFolder*. It is required that the platform model is constructed before invoking the interpreter.

For the same integrability reason, we allowed manual specification of components' rates and/or priorities. The later algorithms of timing and scheduling policy assignments manipulate only those components whose timing and scheduling attributes are not manually specified.

Timing and schedulability analysis interpreter. The timing and schedulability interpreter contains the deadline distribution algorithm and the analyses of both local and global systems. The analysis algorithms work with models stored in *TaskFolder* and *HWFolder*. The deadline distribution algorithm should be invoked first by choosing the *Deadline Distribution* button in the interpreter. The results of released offsets and intermediate deadlines of both tasks and messages are displayed in a separate window.

The timing and schedulability analyses are invoked by *Schedulability analysis* button in the interpreter. One of the three analysis results, individual tasks on a processor, individual messages on a network link, and end-to-end transactions, is displayed at a time. We assumed that the fixed-priority based RTOSs are used in the platform configuration. Under this assumption, all tasks must be assigned some fixed priorities to run. We have implemented three priority assignment algorithms in current version of AIRES: rate-monotonic assignment, deadline-monotonic assignment, and user-defined assignment. In case that the constraints are not all met, a *Refine* button is provided to invoke the automatic refinement algorithms. Further, a *Plot* button is implemented to show the task executions in a Gantt chart.

To facilitate the implementation after the system is analyzed that all constraints are met, we implemented an OIL file generator. The OIL file is the configuration file for OSEKWorks, which defines the hardware platform, software tasks with their properties and allocations, communications, resources, and triggering mechanisms. Given an OIL file and the components, the system can be directly generated using a tool such as WindRiver Tornado, and then downloaded to the target for execution.

Although all algorithms were implemented in AIRES as interpreters that is tightly integrated with the GME environment, the algorithms can still be used individually to perform the desired analysis. We will detail this by showing analysis of a task system generated from Teja tool directly in the evaluation in Chapter 6. In this analysis, the composition check and runtime model generation functions were ignored. Further, at the end of each algorithm, we implemented a *Save Result* option to allow the user store the results in a file for future use.

5.2 Tool implementation for Avionics

The AIRES tool for Avionics was implemented as a standalone program. Different from the Automotive tool implementation, which was based on a common meta-model AIRES and had a fixed, built-in development process, the AIRES tool for Avionics was implemented based on a common exchange file format, called *Analysis Interchange Format* (AIF). The AIF was developed and used within the MoBIES program. AIF is a subset of Embedded System Modeling Language (ESML), and defines only those modeling elements and attributes related to analysis, including component structure and ports, event channels and invocations, configurations, distributions, and execution times of actions.

Given an AIF file generated from ESML models, AIRES extracts system-level dependency information, including event- and invocation-dependencies, and constructs port- and component-level dependency graphs. Various analysis tasks are supported based on these graphs, such as checking for anomalies such as dependency cycles, visual display of dependency graphs, as well as forward/backward slicing to isolate relevant components. It then assigns execution rates to component ports, and uses real-time scheduling theory to analyze the resulting system of real-time task set. If the task set is not

schedulable, the designer can add more processors and allocate components to them with the help of the automated allocation algorithm.

The AIRES tool for Avionics contains the algorithms of dependency analysis, component allocation, and real-time analysis. Since the Avionics applications are mainly modeled with data-oriented components, the analysis is implemented based on Port Dependency Graphs (PDG) translated from the UML-like component diagrams. The AIRES tool provided an opeartion to graphically display the translated PDG, as shown in Figure 5.4. The display was implemented using the Graphviz tool and language developed by AT&T [7].

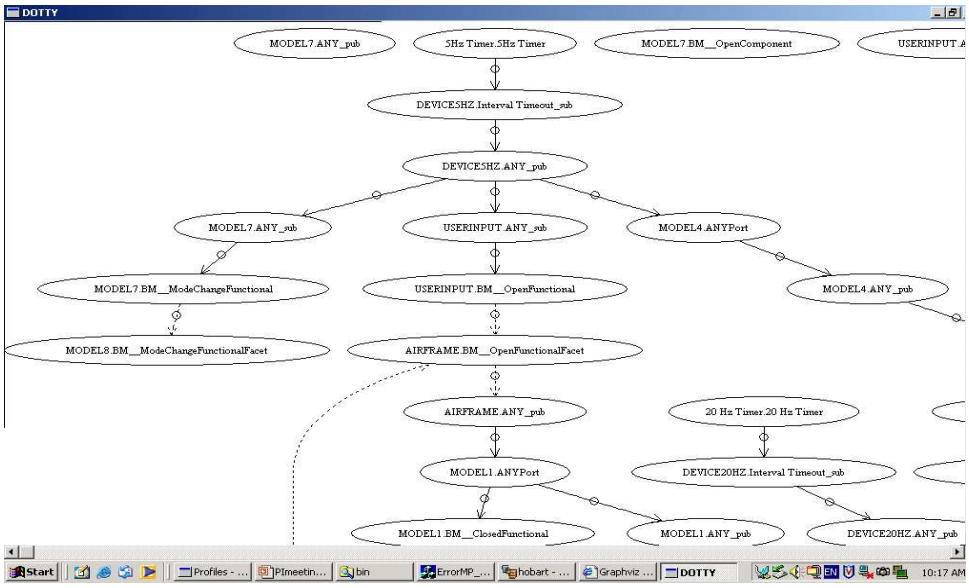


Figure 5.4: A translated PDG graph.

Dependency analysis. The dependency analyses includes event depedencies and invocation dependencies. The dependency analysis should detect potential errors such as cyclic dependencies, events without a consumer, and components not reachable from any timer. The analysis algorithms are invoked when the AIF file is imported into the tool. After the importation completes, the errors are listed in the warning report generated by AIRES tool. Figure 5.5 shows an example display of dependency analysis report in AIRES.

Real-time analysis. The real-time analysis in AIRES for Avionics tool contains the algorithms for local and global schedulability analysis, and rate group assignment. The rate assignment was implemented using an algorithm based on Algorithm 6. In this algorithm, we followed one timer event at a time, traced the events flowing downstream, and marked visited components for assigning the rate of the timer. This design strategy was chosen according to the system characteristics of Avionics applications — every component runs at only one rate. Any component subscribed to multiple events at different rates should be examined, as it could be an error. To help the designer to identify the potential errors, we implemented forward slice and backward slice functions in the analysis. The forward slice starts from a component and graphically shows all components triggered by its output(s) downstream. On contrary, the backward slice starts from an indicated component and graphically shows the upstream

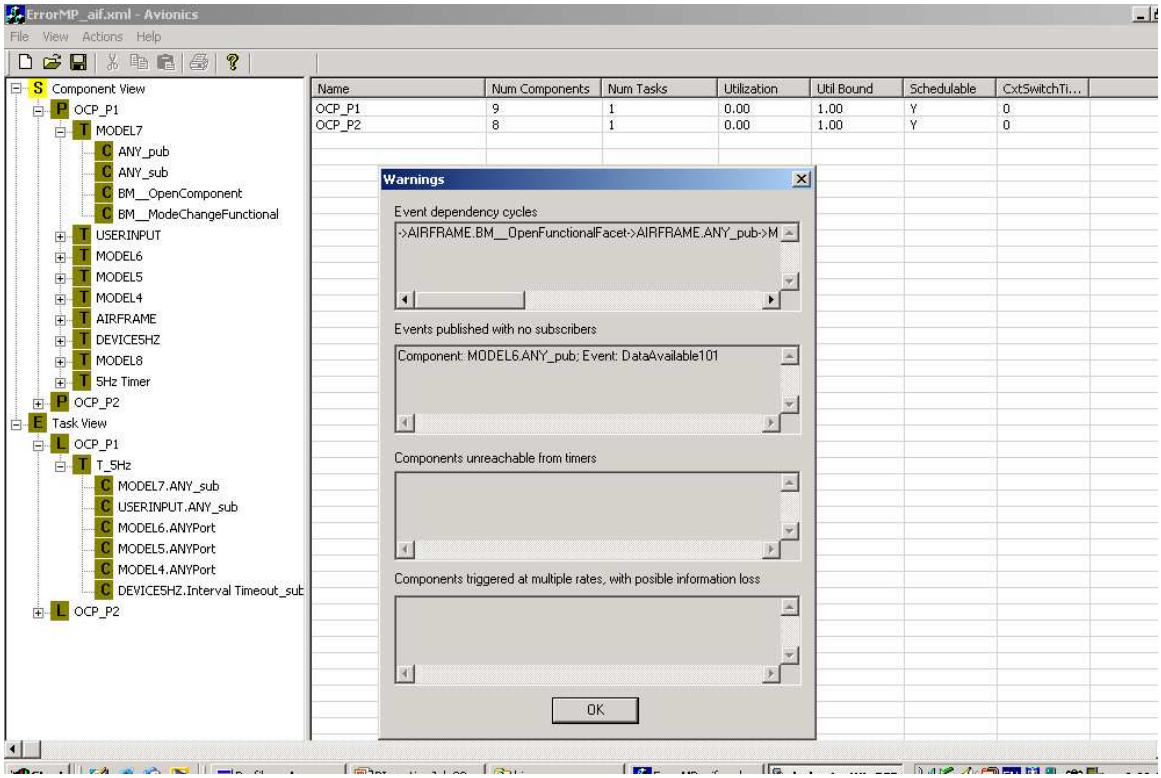


Figure 5.5: Example error report of dependency analysis.

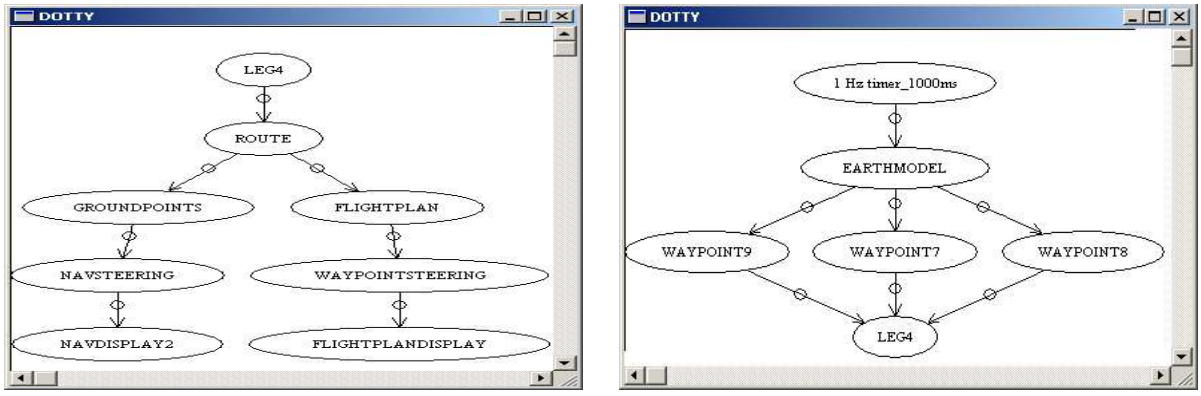
components that trigger it. Figure 5.6 shows an example result of backward and forward slice. By examining the results of these results the design can easily identify the source of conflict rates.

If the rate assignment completes correctly, the assigned results can be used to update the AIF file, which in turn can be reloaded in the modeling tool.

The timing and schedulability analysis was also implemented as a backend operation during the AIF file importation. After the AIF file is imported, the schedulability decision and the worst-case response times for all tasks are displayed in the main information sheet. To perform the analysis, worst-case execution times of the tasks and scheduling policies used in the platform must be given. According to our Avionics OEP partner, the worst-case execution times of components and tasks were obtained through measurements on a real target. The collected timing data was then used to patch the AIF file by appending the execution times to components and tasks. This was done using IIF2AIF, a different tool developed by Southwest Research Institute. For scheduling policy selection, we implemented only rate-monotonic schedulability ananlysis and chose the rate-monotonic priority assignment as this is the only policy used in Avionics systems.

Besides the local schedulability analysis, we implemented the global analysis, called *end-to-end timeline*, in the AIRES tool for Avionics. The end-to-end timeline analysis was designed to verify timing constraints satisfaction of transactions (called system threads in Avionics) that involving multipe tasks on different processors. This function is useful to check timing errors such as frameoverruns. The results can be graphically displayed as shown in Figure 5.7.

Component allocation. The implementation of component allocation in the AIRES tool for Avionics was based on the algorithms in Section 3.2.1. Different from the AIRES tool for Automotive, there was



(a) Forward slice.

(b) Backward slice.

Figure 5.6: An example of forward and backword slice.

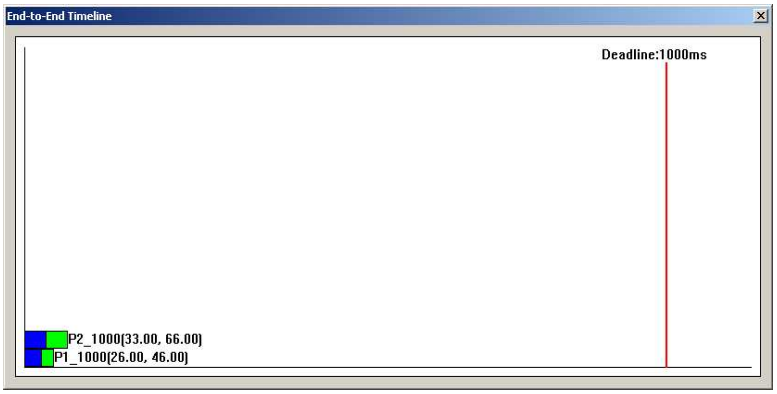


Figure 5.7: An example end-to-end timing analysis result.

no process for task formation afterwards. This is due to the domain specific design methods in Avionics. With such a method, one thread dedicated to a rate is pre-constructed on all processors. Therefore, the task/thread containing the component is determined after the rate and the processor of the component are assigned.

The AIF file that contains the model information must be loaded in the system first. The AIF file must be patched with worst-case execution times for all componnets (if this isn't done the worst-case execution time of the component is assumed to be zero). There are 2 considerations that can be used for component allocation: the maximum number of components on each processor and different allocation strategies. The maximum number of components specifies the limitation of the processor. The allocation strategies include first-fit, load-balance, and minimization of communication. The number of processors in the system, on the other hand, can not be modified in AIRES tool. It can only be changed in a modeling tool and then converted to an AIF file.

The generated allocation can be used to update current AIF file, which in turn can be imported back to a modeling tool. If the models are updated with AIRES component allocation, a new configuration model will be created in the modeling tool.

Chapter 6

Evaluation Results

We have designed and performed a set of experiments to evaluate the effectiveness and applicability of the AIRES tool. We evaluated both individual algorithms and the integrated tool chain. The evaluation of algorithms focused on the computation complexity. The integrated tool chain evaluation focused on the integrability with other tools and overall effort savings in the development process.

6.1 Evaluation of analysis algorithms

Among the developed algorithms, the algorithms for composability analysis and event/invocation analysis address the functional design issues, and were evaluated in the tool chain integration experiments. Similarly, the algorithms for timing attribute assignment, task formation, and schedulability analysis are either straight forward or use a simple revised version of some traditional algorithm to suite for the tool integration. These algorithms were also evaluated mainly in tool chain integration experiments to demonstrate the overall effort savings in the whole development cycle. So our individual algorithm evaluations focus on the component allocation algorithms, and task dependency resolving.

6.1.1 Component allocation algorithms

The evaluations of the component allocation algorithms aimed to understand the computation complexity of the algorithm and the quality of the generated results. To this end, we chose the number of steps to generate a result as the metrics for computation complexity. The steps performed to generate a result indicates the the time for the algorithm to generate a result, thus is usually used to evaluate the scalability of the algorithm. To evaluate the quality of the generated results, we compared the results generated by the algorithm with the optimal results generated by an exhaustive search. Since our algorithm uses heuristics to find a solution quickly, the results may not be optimal. The smaller the difference between the results of our algorithm and the optimal solution, the better quality of results our algorithm generates.

To evaluate these allocation algorithms, we designed our experiments by randomly generating a set of directed acyclic graphs. Each graph represented a transaction in the structural model. Table 6.1 lists the system configuration parameters in the design of experiments.

	# trans/model	# comp/trans	# conn/comp	# inputs/trans	# outputs/trans	# processor
value range	$5 \sim 100$	$10 \sim 1000$	$0 \sim 6$	$1 \sim 10$	$1 \sim 10$	$1 \sim 40$
workload (%)	-	$0.001 \sim 0.1$	$0.01 \sim 0.2$	-	-	$0.4 \sim 1$

Table 6.1: Parameters for random system generation.

In Table 6.1, the values of workload defines the workload introduced by each component or connection.

The workload introduced by a component was computed as the component's execution time divided by its invocation period. The workload of a connection was computed as the size of messages sent in a unit of time divided by the bandwidth reserved for the communication. Differently, the workload of each processor in the table defines the utilization bound of the processor, which total workloads of the components allocated on it should not exceed.

We first evaluated first-fit, load-balance, and communication minization. We used the number of steps (iterations) each algorithm performed before a solution was found for performance metrics of the allocation algorithms. Such performance was evaluated using various graph size, which were the number of components in the graph. The performance metrics reflected the scalability of the algorithms. Figure 6.1 showed the steps required to generate a solution with each algorithm.

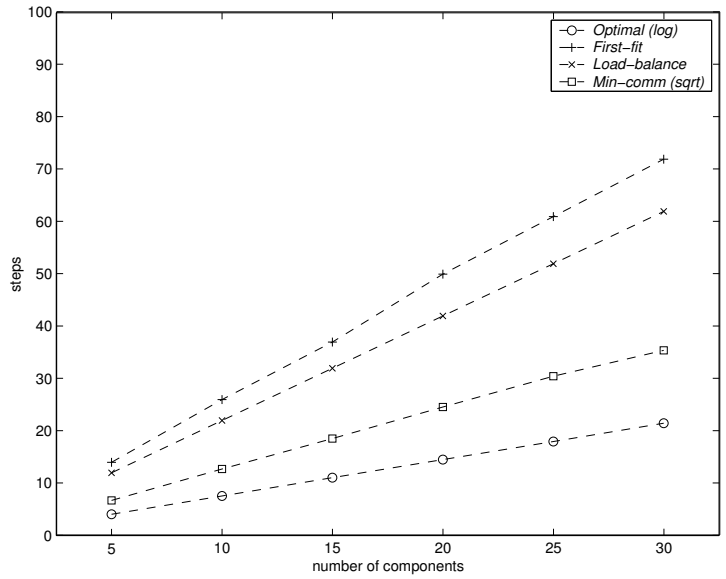


Figure 6.1: Scalability of the algorithms with different graph size.

The results shown in Figure 6.1 were generated with fixed with 2 processors. The workload for each measured case was between 1.2 to 1.5. This was chosen to ensure that there were components allocated on both processors. To make the results fit in the diagram, we took the natural log of the result for optimal and a square root of the result for communication minimization. As can be seen, the number of steps needed for the optimal solution was exponential, while it was polynormial for communication minimization, and it was linear for first-fit and load-balance algorithms.

To evaluate the quality of the algorithms, we compared the solution of our algorithm with an optimal solution using a small size graph. The small size graph was chosen to guarantee an optimal solution found within a reasonable time during the experiment. We limited the number of components in the graph to be 30 for the experiment. This was based on experiments that showed the execution time of the optimal algorithm with components greater than 30 took over 10 hours to generate a result. To find an optimal solution, we implemented an exhaustive search algorithm based on computing permutation of allocations. The quality of the algorithms were shown in Figure 6.2, 6.3, and 6.4. Since we would like to keep the similar workload across the experiments with different numbers of components, the worloads introduced by each components was decreased as the number of components increased in the experiments.

All results showed that the differences between the optimal solutions and the heuristic results were

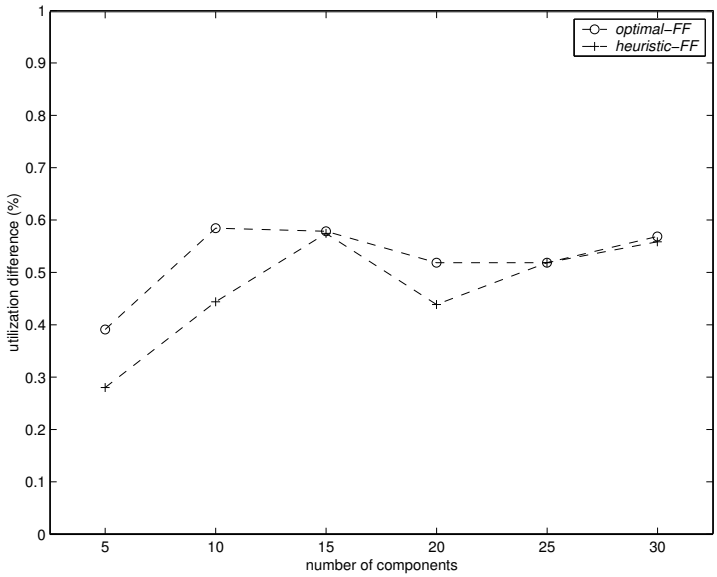


Figure 6.2: Quality of the first-fit algorithm with different graph size.

close. For first-fit algorithm, the larger the difference between different processors, the better the results are. The differences between optimal results and heuristic results tend to reduce as the number of components increased. This scenario was in fact a result of the decreased utilization of components. As the workload of each component becomes small, it is easier to fully utilize a processor than using components with larger utilizations. Similarly, the difference between optimal and heuristic algorithms for load-balance were also close. In this case, the smaller the difference is, the better quality of the algorithm is. The optimal results for load-balance is almost constant zero. The heuristic algorithm resulted in the optimal results for even number of components and a small difference for odd number of components. We observed that the difference became smaller as the workload of components decreased. This may be caused by workload bounds introduced during the random graph generation. Such bounds enforced the component workloads to be similar. Therefore, it is easier to balance the workload on two processors when the component number is even. The difference between optimal and heuristic algorithms for communication minimization also was shown to be small.

Besides the graph size, the algorithm complexity depends also on the platform configuration. We performed a set of experiments with different number of processors. The number of processors were chosen from 2 to 7. The number of components were fixed to be 20 for comparing with optimal solutions. Any configuration with a processor number greater than 8 took an extremely long time to complete. The steps for our algorithms and optimal one are showed in Figure 6.5, and the quality of our alogirhtms compared with the optimal solutions were given in Figure 6.6, 6.7, and 6.8.

Again, we converted the steps for the optimal algorithm to its ln value to make it fit in the diagram. Similarly, a square root of the steps of communication minimization was used. The results showed that the complexity is exponential as the number of processors increases for the optimal algorithm, is super-polynormial for our communication minimization algorithm, and is linear for both our first-fit and load-balancing algorithm. The quality of the algorithm was represented using cumulative utlization difference, which was computed as follows:

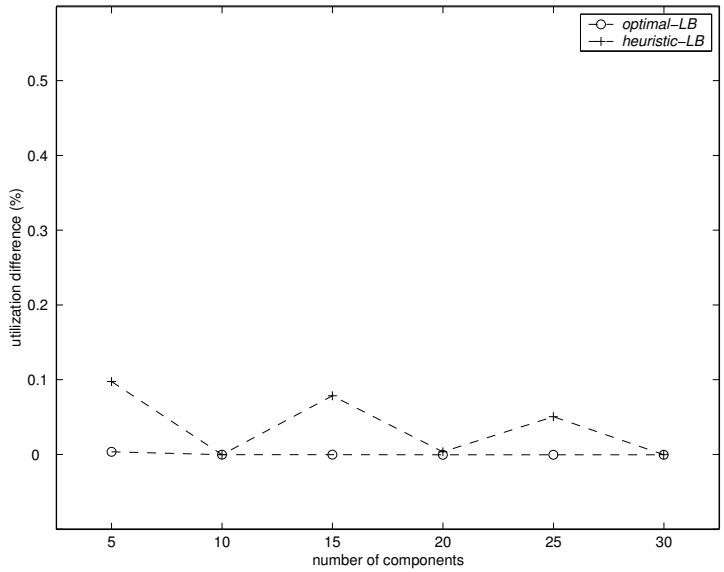


Figure 6.3: Quality of the load-balance algorithm with different graph size.

$$U_\sigma = \sum_{i=1}^{n} \sum_{j=i+1}^{n} (U_i - U_j)$$

where i and j are indices of processors that are sorted in decending utilization order.

The quality of our algorithms generated almost the same results as the optimal ones for both first-fit and load-balance algorithms. The difference between the results of our communication minimization algorithm and the optimal algorithm was within the same bound as that of the experiment with different graph sizes. This indicated that the graph size has more significant impact to the algorithm quality than the number of processors for first-fit and load-balance algorithms. On the other hand, the effects of the number of processors and graph size are equivalent for communication minimization algorithm.

The experiments with optimal solutions only ran in a small scale due to the complexity of the optimal algorithm. To evaluate the scalabity of our algorithms in a large scale with large number of components, we generated graphs with component number from 100 to 1000. These experiments included the combinational allocation algorithms with considerations of both computation resource and communication resources. In this set of experiment, the number of processors was fixed to 4. The total system workloads were assigned randomly from 2.3 to 2.7. Figure 6.9 shows the number of steps for each algorithm to generate a solution.

It can be seen that the complexity of first-fit and load-balance algorithms are still linear, while the rest of the algorithms are all polynormial. All algorithms generated a solution with 30 minutes on a P3 800 MHz machine. We also did experiments with some graphs containing 5,000 $\sim$ 10,000 components, and experienced from 4 $\sim$ 10 hours before a solution was found on the same machine.

We developed two heuristic for communication minimization algorithm with load-balance. One was a simple heuristic that merges the components connected with the great cost link (H1). Another was the one proposed in [1] that summing the costs of all links of a component, and selecting the component with the highest cost sum to merge (H2). Similarly, we fixed the number of processors to be 4, and

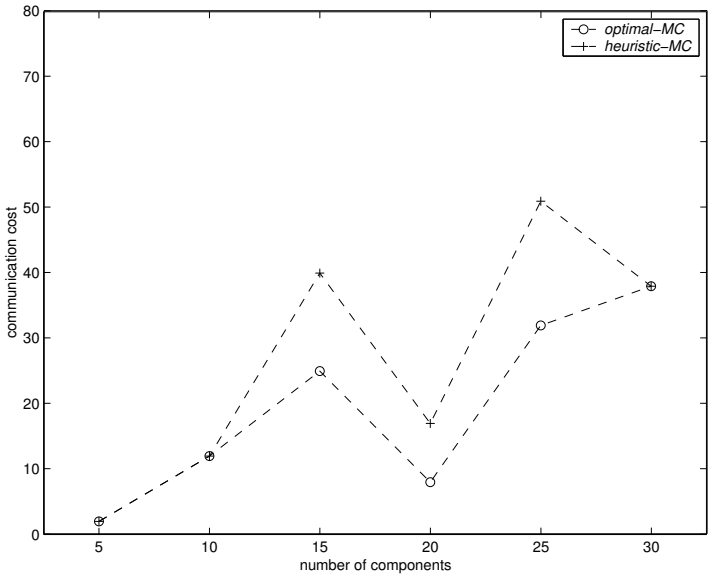


Figure 6.4: Quality of the communication minimization algorithm with different graph size.

assigned the system workload between 2.3 to 2.7. Figures 6.10, 6.11 and 6.12 show the experiment results.

The results showed no big quality differences between heuristic H1 and heuristic H2. However, according to the results in Figure 6.10, heuristic H1 has much lower complexity than heuristic H2. Therefore, we recommand using H1 as the main heuristic for components' allocation with communication minization.

6.1.2 Dependency resolving algorithm

The experiments in this section were designed to evaluate our approach for the dependency resolving algorithm. Since the goal of this algorithm is to break a dependent task set into an independent task set so that a simple, polynomial time TAS algorithm can be applied, we need to select a TAS algorithm in the evaluation. The TAS algorithm was applied equally to both the dependent task set before transformation and the independent task set after transformation in order to show the performance differences. Without loosing generality, we chose the first-fit algorithm for task allocation. Further, we assigned priorities of tasks using deadline-monotonic assignment with a higher priority for the with a tighter deadline. The scheduling algorithm used in the experiment was the holistic scheduling approach [3]. The holistic scheduling analysis uses the same time-demand function as our implemented local and global schedulability analysis algorithms in the AIRES tool. The only difference is that the holistic scheduling analysis considers the release jitters of tasks. [1]

Our algorithm for polling rate derivation partitioned end-to-end timing constraints over individual tasks or composite tasks by using two well-known deadline-distribution heuristics: *pure slicing* and *normal slicing approach* [14]. The pure slicing approach distributes slack in the deadline according to the number of tasks in the directed acyclic graph, while the normal slicing approach distributes the deadline to component tasks in proportion to their execution times. Only *local clustering* is used in comparisons to evaluate the overhead reduction methods.

[1] Since we were only interested in the relative performance of the shared-buffer approach, the choices of algorithms will not affect the comparison results.

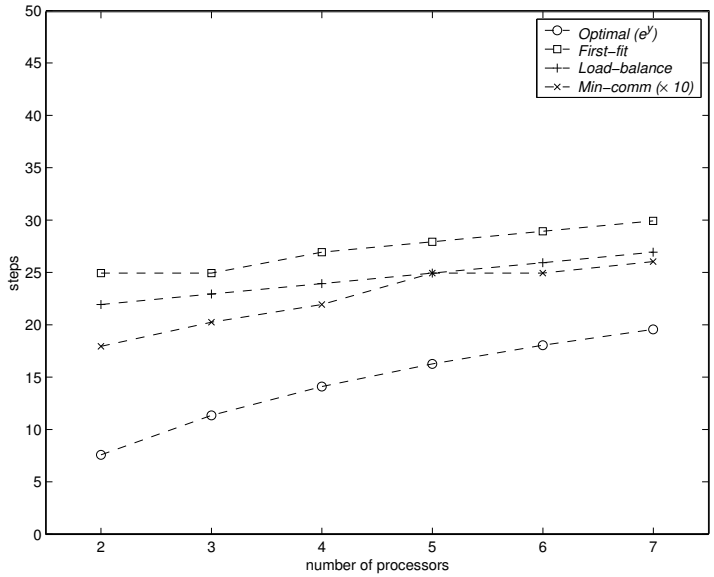


Figure 6.5: Scalability of the algorithms with different numbers of processors.

The simulated system we exercised contains a set of tasks whose number ranges randomly from 60 to 300, partitioned into $10 \sim 50$ groups, with each group consisting of $4 \sim 8$ dependent tasks.

Table 6.2 summarizes the selected algorithms and system parameters in the simulation.

Configuration parameters	values
Deadline-distribution	Pure slicing, Normal slicing
Task clustering	Local clustering
Task allocation	First-fit bin packing algorithm
Scheduling	Static priority pre-emptive scheduling
Average out degree of each task	2
Average in degree of each task	2
Execution time of each task	$5 \sim 10$ ms
Period of a dependency graph	$50 \sim 150$ ms
Slack for a dependency graph	$2.5 \sim 7$ times the sum of execution times of tasks in the graph
Polling overhead	$5 \sim 10\%$ of single task execution times

Table 6.2: Algorithm selection and graph characteristics in the simulation system.

In the simulation, we first constructed several task graphs with properties assigned according to Table 6.2. Then, they were transformed into independent tasks using the shared-buffer approach. Algorithm PP_D was executed to iteratively generate task allocation and scheduling, and provided the number of processors used to schedule the given task set, processor utilizations, polling overheads, and the number of iterations to generate a schedulable task set. This process was repeated 3 times with different random seeds and an average was taken on the obtained values.

Experimental results:

To find a schedulable task allocation, the total number of schedulability checks performed in each iteration by the first-fit TAS is of order $\Theta(n*p)$, where n and p are the number of tasks and the number

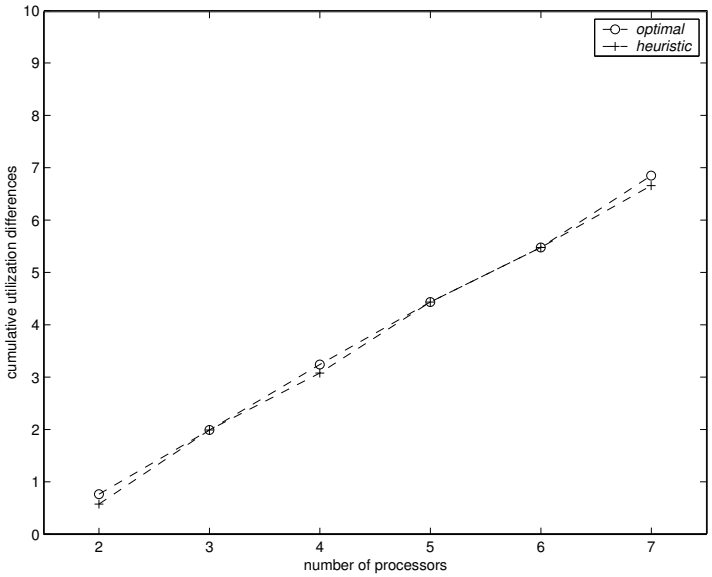


Figure 6.6: Quality of first-fit algorithm with different processor number.

of processors, respectively. The total number of checks performed during the execution of Algorithm PP_D depends on the number of iterations required to converge to a solution. In each iteration, any task that does not satisfy Eq. (3.2) has its polling period reduced using Eq. (3.4). Since the polling period of a task can only decrease with each iteration until the task can be either satisfactorily scheduled, or below a specified threshold when the algorithm gives up, the number of iterations in which each task has its polling period revised, is bounded. This sets an upper limit to the number of iterations that the algorithm can perform, and this limit is linear in the number of tasks for a given slack. The experimental results are shown in Figure 6.14 and 6.13,

The number of iterations required to allocate tasks on processors with a different number of tasks in the task set is given in Figure 6.14. In this experiment, we fixed the slacks in all dependent-task graphs to a constant of 7 times of the execution time for all tasks. The results in Figure 6.14 showed that the number of iterations increases almost linearly with the number of tasks. Irrespective of the number of tasks in the system, the number of iterations to generate a solution is bounded around 20, even in a graph of 250 tasks. This indicates that the TAS problem for a large task set can be solved in a short time using our approach.

The results on the efficiency of using shared buffers are plotted in Figure 6.13, showing the number of processors required to make the task set schedulable using the first-fit algorithm. We compare the number of processors returned by Algorithm 9 that makes the system schedulable with that returned by the first-fit TAS on the same system without inter-task dependencies. All tasks in the system without dependencies share the same period and deadline as that of the original system with dependencies. We set the maximum number of tasks in this experiment to 220 tasks.

The results in Figure 6.13 show that without clustering, allocation algorithms using the polling method perform worse when tasks have low slacks in their deadlines. This is because the polling rate for tasks gets higher after transformation when the slacks are low, thus wasting more processor time for polling. But for larger system slacks, the processors returned by the polling approach asymptotically met that of scheduling the system without any inter-task dependencies. The polling rates for dependent tasks decrease as the system slacks increase. In such cases, truly independent tasks require fewer

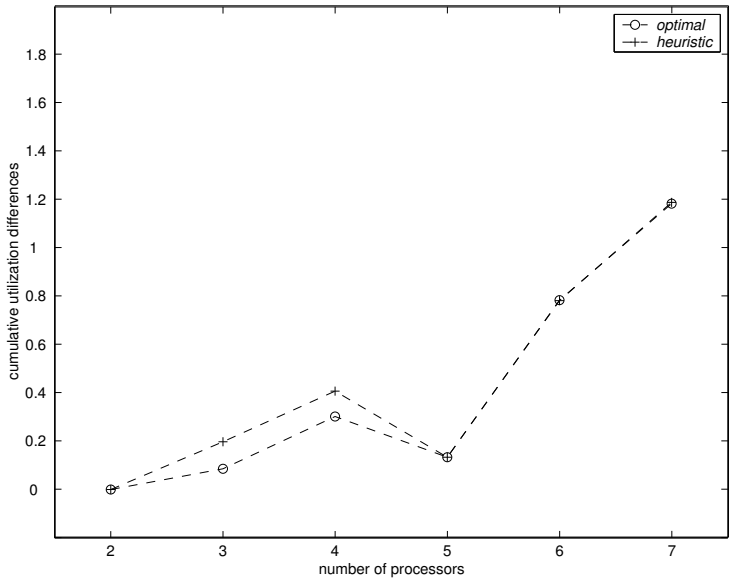


Figure 6.7: Quality of load-balance algorithm with different processor number.

processors for them to be schedulable. This implies that our approach is more useful when tasks have more slack and longer deadlines, which is commonly the case in large-scale distributed systems.

Figure 6.13 also shows the efficacy of task clustering in terms of polling overhead reduction. Even at low system slacks, if the local clustering was used, the number of processors required to schedule the task system was only marginally higher than that required to schedule the same system without task dependencies. This indicates that task clustering was indispensable to the shared-buffer approach.

Figure 6.15 shows the polling overheads before and after task clustering. The polling overhead is defined as:

$$\text{polling overhead in } \% = \frac{\dfrac{\sum_{p \in P} \sum_{T'_j \in TS_p} \dfrac{pollexec_{T'_j}}{poll_{T'_j}}}{|P|}}{\dfrac{\sum_{p \in P} \sum_{T'_j \in TS_p} \{\dfrac{pollexec_{T'_j}}{poll_{T'_j}} + \dfrac{e_{T'_j}}{p_{T'_j}}\}}{|P|}}$$

where P is the set of processors, TS_p the set of tasks allocated on p, $pollexec_{T'_i}$ the execution time for polling T'_i, $poll_{T'_i}$ the polling period of T'_i, $p_{T'_i}$ the invocation period of T'_i, and $e_{T'_i}$ the execution time for data processing of T'_i. The numerator in the above equation denotes the polling utilization averaged over all tasks and processors. The denominator in the equation denotes the sum of polling and task utilizations averaged over all tasks and processors.

From Figure 6.15, we observed that the overhead without task clustering was $2 \sim 3$ times higher than that with local clustering. We also observed that the polling overheads decreased as the system slacks increased. This was also caused by lowering the polling rates of tasks with increased slack. With task clustering, the polling overheads were reduced and thus more acceptable as they were around less than 15%, even at low system slacks.

We also observed that the normal deadline-distribution algorithm tends to introduce smaller polling

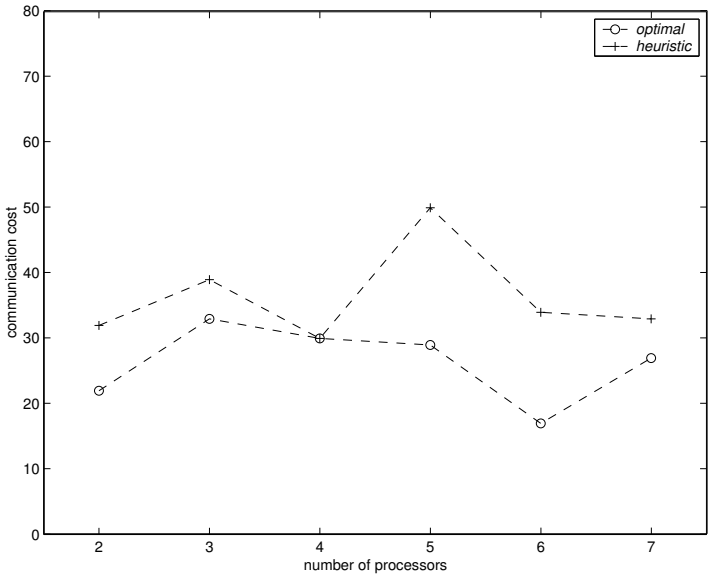


Figure 6.8: Quality of communication minimization algorithm with different processor number.

overheads, use fewer processors, and assign lower polling rates for dependent tasks than the pure deadline distribution when task clustering is not used. When task clustering was used, both the normal and pure deadline-distribution algorithms tend to generate similar results on these metrics.

As the TAS for precedence-constrained periodic tasks in a distributed sysem is NP-complete, existing heuristics for determining the satisfiability of a task allocation can be very complex [32], or involve generating the entire schedule up to the planning cycle — LCM (least common multiple) of periods — of all tasks on each processor [22, 21]. These approaches are computationally very intensive and can take a very long time. Since such timing satisfiability checks are invoked at each step of any TAS algorithm, the running times can be very large even for moderately-sized task sets.

In the polling method, the global end-to-end timing constraints are broken into individual timing constraints on component tasks. Each task is made independent of others using shared buffers. Consequently, determining the satisfiability of a given deadline allocation to tasks only involves schedulability check on each processor. Since such checks for independent periodic tasks can be done accurately and very fast, TAS using the polling method is much faster than the usual non-polling approach.

To evaluate this, we compared the running times for TAS using the polling method (polling TAS) with TAS not using polling method, but allocating and scheduling based on dependencies (regular TAS). For polling method, we used normal deadline distribution algorithm. For regular TAS, we used a simple depth-first search (DFS) algorithm that terminates after finding the first solution that satisfies the timing constraints. To determine timing satisfiability, we used a simple algorithm that determines the response time of each task using holistic analysis [3], and then determines the end-to-end response times using the response times of individual component tasks. To see how fast this simple check is, we determined the time taken for the satisfiability tests for an example task set in [32], on a sun workstation. Our simple check took 0.0019 second whereas the iterative method in [32] took 0.13 second. Using this simple check for DFS TAS, we measured the times taken to allocate and schedule 15 task graphs with characteristics as given in Table 6.2, on a 1.2GHz AMD Athlon processor with 256MB RAM. The time taken for both approaches for systems with different slacks is shown in Table 6.3.

Even with a simple satisfiability check, the DFS algorithm takes a much longer time than the polling

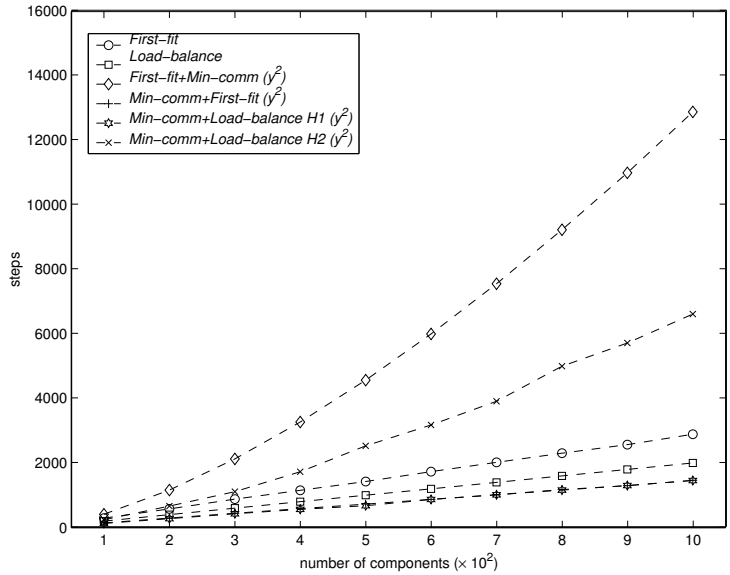


Figure 6.9: Scalability of algorithms with large graph size.

Approach	Time to find a solution (sec) (slack=5.0)	Time to find a solution (sec) (slack=7.0)
Polling method	1.5	1.3
DFS method	2334	2111

Table 6.3: Polling method and regular TAS

method. We also found that since DFS is not an optimal algorithm, it takes more processors than the polling method with first-fit bin packing allocation to schedule the tasks.

A more rigorous timing check like in [32], and to optimize TAS for a metric like load balancing, or to minimize processors, would increase the time for TAS significantly when dependencies are considered. With the polling method, however, it is easier and much faster to perform TAS to optimize for metrics like load balancing, or minimizing processors owing to its simpler timing satisfiability checks and task independence.

However, because of the overheads associated with the polling method, its solutions may be inferior to optimum solutions obtained directly. To evaluate how the polling method compares with an (optimum) regular TAS, we compared the number of processors required to schedule 9 task graphs using the polling method and an exhaustive search TAS algorithm. The characteristics of the task graphs are given in Table 6.2. Table 6.4 shows the number of processors required and their average utilizations.

For a system slack of 2.0, due to its overheads, the polling method requires, on average, an extra processor to schedule the task set. However, when the system slack is higher at 5.0, the polling method requires the same number of processors as the regular TAS method. As described and evaluated above, a higher slack in deadlines decreases the polling overhead, making the performance of the TAS algorithm using polling closer to the optimum.

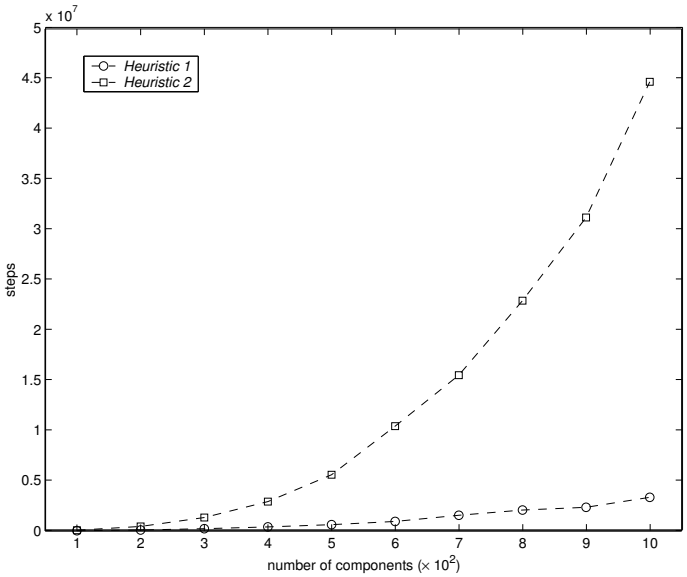


Figure 6.10: Scalability of heuristic.

	feasible solution (slack=2.0		feasible solution (slack=5.0	
Approach	# processors	average utilization	# processors	average utlization
Polling method	6.67	0.82	6	0.80
Optimum TAS	6	0.76	6	0.76

Table 6.4: Polling method and optimum TAS algorithm

6.2 Evaluation with tool chain integration

The tool chain integration experiments were performed to investigate the integratability of the AIRES tool. This is the main object of the MoBIES program. The experiments of integrations were done with assistences from our OEP partners in the MoBIES program. The experiments included taking application models from our OEP partners, translating them into the models compatible with AIRES modeling language, carrying out the analysis, and either feeding back the analysis results to the modeling tool or tagging the models with the analysis results. Since the application domains of automotive and avionics are different, and our AIRES toolkit was applied differently to these two domains, we discussed the integration experiments separately in the following sections.

6.2.1 Evaluation with Automotive OEP applications

The tool chain integration experiments for Automotive involved two applications: the Electronic Throttle Control (ETC) and Vehicle-to-Vehicle (V2V) application. The ETC application experimented every aspect of AIRES toolkit through the end-to-end design process, including component modeling, functional model design, runtime model generation, system analysis, and code generation. The V2V application experimented the case where the AIRES toolkit is used only for analysis in the end-to-end design process. The evaluation metrics of the evaluations included: (i) the AIRES tool can take models output of upstream tools and generate information for downstream tools, and (ii) the AIRES tool generated results that can help to make better design.

The experiments were carried out on a testbed with properties shown in Table 6.5.

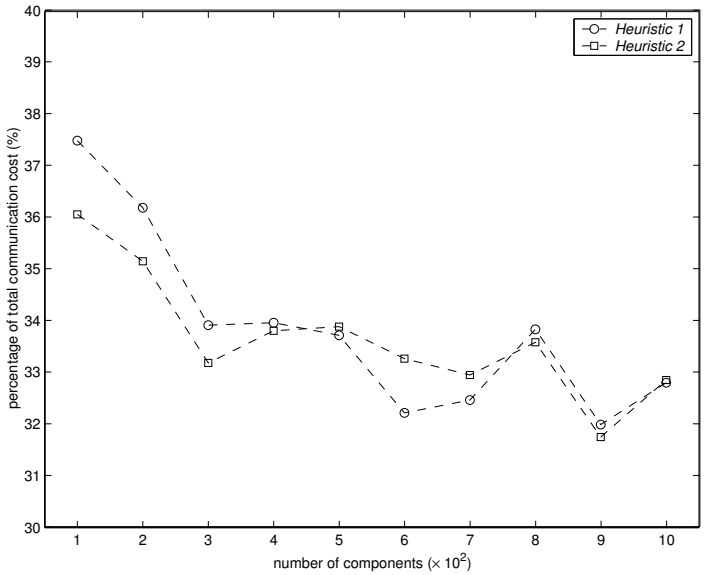


Figure 6.11: Communication cost of solutions using different heuristic.

Experiment	HW	OS	upstream tool	downstream tool	model envrionment	AIRES tool
AFR+ETC	Pentium 850 MHz 256 MB	Windows 2000 SP6	Simulink Stateflow	WindRiver Tonado 2	GME3 r3.3.28	v2.5
V2V	Pentium 850 MHz 256 MB	Windows 2000 SP6	Teja	—	GME3 r3.3.28	v2.5 sched.dll

Table 6.5: Environment for the experiments using Automotive application.

The experiment process of ETC application was as follows. The model we used was AFR control with ETC simulink model release version 1.0 given by Automotive OEP. The provided models included an ETC monitor subsystem, an ETC manager subsystem, an ETC servo-control subsystem, an Air-Fuel-Ratio control (AFR) subsystem, and a SFP subsystem. These subsystems interacted through data flow. The activations of these subsystems were defined in a set of Stateflow diagrams with invocation periods were given. Since the given ETC-AFR model was limited on timing and schedulability model, we imposed the needed information artificially and made some limited changes of the models' timing information to test more functionality of AIRES tool. Specifically, we kept all rates of the original model, and the execution times measured at our lab using the OEP provided C code. Then we performed the component allocation and task formation to generate a runtime model. The timing and schedulability analysis was carried out with the generated runtime models afterwards, and provided the analysis results. After the model passed the analysis, we generated the configuration file so as to generate the target code.

We first translated these Simulink/Stateflow models into AIRES component models in GME. These models were stored in Simulink/Stateflow as reusable components. The component hierarchy in Simulink/Stateflow was maintained in the translated component models. Components at different hierarchies can be reused

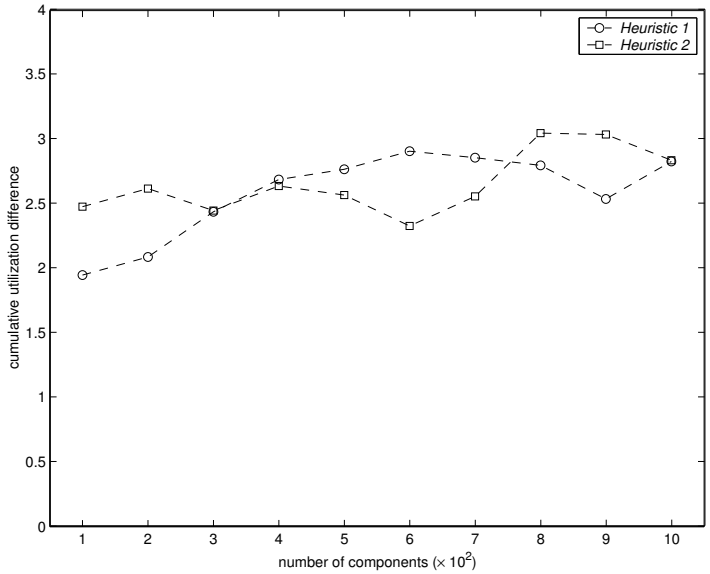


Figure 6.12: Utilization of solutions using different heuristic.

equally in the application construction. We recorded the time it took to import the models and complete the analysis. Table 6.6 shows the results of this experiment steps.

operation	Application model	Simulink		Stateflow		operation time (sec)
		# elements	# hierarchy	# elements	# hierarchy	
Model import	etc	1400	6	243	7	170
	afr	793	7	13	4	67
	sfp	89	5	5	2	30
Signal import	monitor	25				3
	manager	43				8
	servo	62				14
	afr	14				4
	sfp	6				1
Signal check						2

Table 6.6: Experiment results of composition check with AFR+ETC.

The number of elements in the each model in Table 6.6 includes all modeling elements such as input and output points, blocks, S-functions, etc. We include all elements at different modeling hierarchies in the model because each of these elements are treated as a unique instance and must be recreated in the component repository during the translation, regardless that there might be some duplication in the original model. The level of modeling hierarchy also affected the time of the translation. The results showed that the number of elements in the model affects the translation time significantly. The signal importation time depended on both the number of the signals and the data types of the signal. The algorithm detected all mismatched signals in the specification files. After correcting them, we obtained a model without functional error.

The components were allocated by the AIRES tool automatically after the functional check. In this

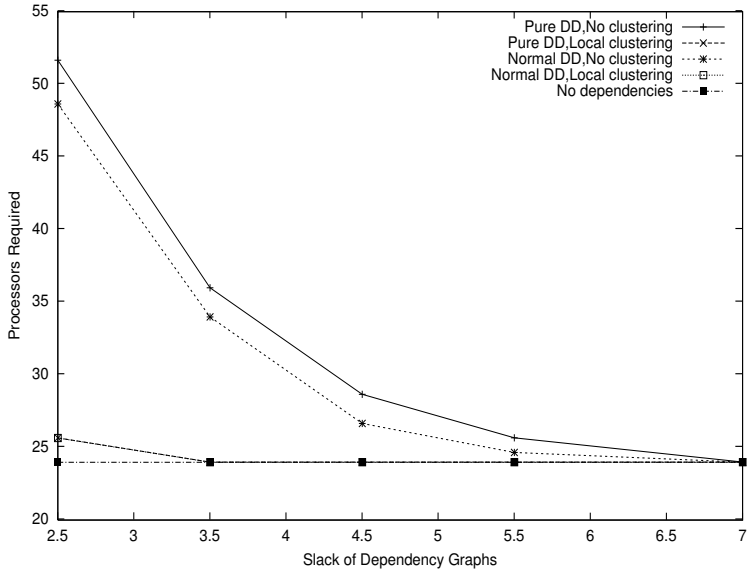


Figure 6.13: Number of processors required vs. slack

step, we manually created a platform with 2 MPC processors. The platform used by OEP to evaluate the final code was running OSEKWorks on MPC 555 and HC08 connected via a serial cable. Since only OSEKWorks on MPC was measured in our lab, we replaced HC08 by another MPC to be able to obtain quantitative results in the analysis. The results of AIRES component allocation showed that all components can be allocated on one processor. Five tasks were generated according to the propogated rates.

The 5 tasks were created with their timing constraints assigned. In this experiment, we assumed the deadlines of each task are equal to their periods. The analysis was performed with the tasks' priorities assigned using rate-monotonic priority assignment. To make the evaluation consistent with OEP's target, we manually allocated the tasks on two processors instead of on one according to AIRES tool. With this setup, the analysis showed that the timing constraints can all be satisfied. Table 6.7 showed the results of AIRES timing and schedulability analysis.

task	period (ms)	wcet (ms)	priority	wcrt (ms)	resource consumption	# preemptions
etc manager	20	0.1	2	2.2	0.005	2
etc monitor	30	0.3	1	2.66	0.01	3
etc servo	3	1.2	4	1.32	0.4	0
afr	4	0.4	3	1.88	0.1	1
spf	1	0.1	5	0.1	0.1	0

Table 6.7: Timing and schedulability analysis results of ETC+AFR.

The results in Table 6.7 included all the overheads such as timer overhead and scheduling overhead of OSEKWorks. The resolution of the clock was chosen as 1 millisecond for tasks of ETC and AFR. The SPF task was assumed to be allocated on a different processor (using a co-processor) running without any OS overhead. Since the SPF task is triggered by the crank shaft, the SPF executes aperiodically. The period for SPF was therefore assigned to be its minimum interval invocation time. Further, all priorities assigned in this experiment were global priorities.

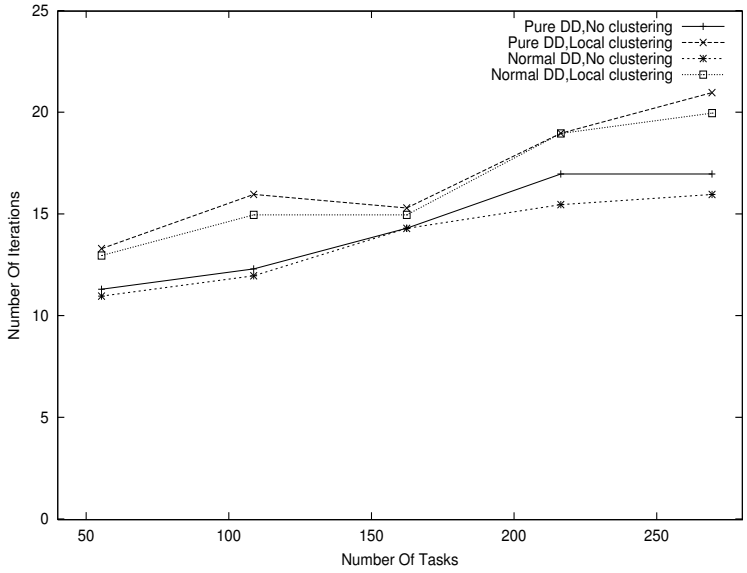


Figure 6.14: Number of iterations vs. number of tasks.

The final target system was generated after all timing and resource constraints were met. The final runtime model was then outputted to an OIL file. The final system was generated by using the OIL file and components' code in WindRiver Tornado 2 — a system target generation tool for an MPC target. Although the target code was generated successfully, we could not experiment it on the target processor since some of the initial model parameters were not available.

The experiment of V2V was to evaluate and test the integratability of AIRES tool by using it only at certain phase of development for certain analysis. In this experiment, we chose the timing and schedulability analysis as the function for tool chain integration. The evalaution metrics included efforts required to using the AIRES timing and schedulability analysis in the tool chain, and the accuracy of the analysis results.

The V2V model was modeled in the Teja tool. To use the AIRES tool for analysis, a runtime model is essential. The system-level tasks and their interactions were provided by OEP in a set of interaction diagrams. All the scheduling properties of the task set including average and worst-case execution times, periods, and priorities were given. Since we did not have any translator developed to convert a Teja model to AIRES, we manually recreated the task graph according to the diagrams given the OEP. The creation included both task set and platform. The V2V application ran on a platform consisting of 2 networked Pentium machines with QNX operating system. The tasks were running as QNX processes. We applied the 2 priority-based scheduling policies in the V2V experiment: *rate-monotonic assignment* and *user-defined assignment*. These two priority assignments were chosen to evaluate the priority assignment in the initial system configuration. Since RM is an optimal assignment for indepenendent tasks with deadlines equal to their periods, the smaller resource useage differences between using given priorities and using RM are, the better the initial system configuration is.

Tables 6.8 and 6.9 show the analysis results of tasks on processor P1 and P2 using RMS. In this experiment, the user-defined priorities were ignored, and the algorithm generated the priorities of tasks according to their periods. A larger number in the tables represents a higher priority.

The analysis results provided the worst-case response times, resource consumptions (as utilizations for computation resource only), and experienced a number of context switches during the worst-case

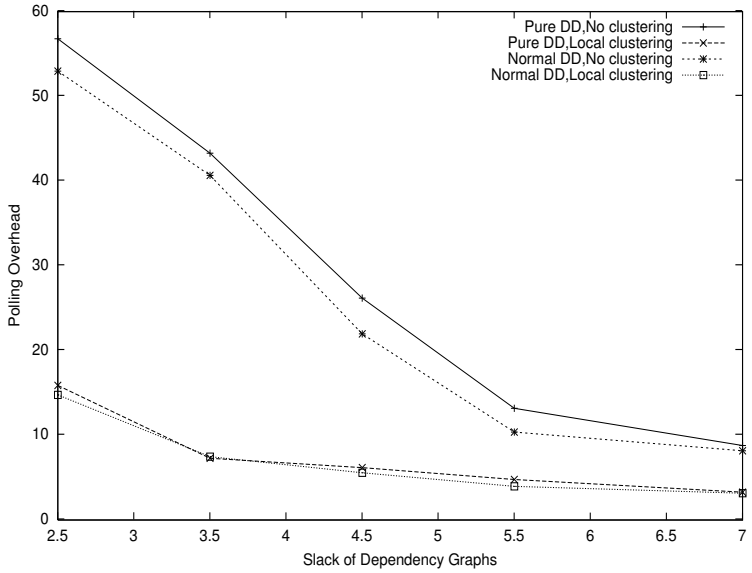


Figure 6.15: Simulation results for polling overheads.

executions. As can be seen from the results, all tasks had the worst-case response times less than their periods. The analysis took into account the overheads and resource consumptions of QNX. So the tasks on both processors are schedulable under RMS. The computation resource consumptions for application tasks, timing service and scheduler on P1 were 23.15%, 0.56%, and 0.0015%, and on P2 were 0.43%, 0.56%, and 0, respectively. This indicates that the workloads on these two processors were not balanced. The scheduler overhead on P2 was extremely small, and was rounded-up to 0. The overheads of the timing service are the same on both processors due to we used the same timer resolution.

We then checked the tasks' response times and the system schedulability under user-defined priorities. The priorities assigned to each task were given in V2V documentation [8]. The analysis results are presented in Table 6.10 and 6.11.

According to the analysis, the task set on processor P1 was unschedulable. Specifically, the worst-case response time of task *veh_lat* was 4.9 ms, which was greater than its period 2 ms. This was mainly caused by the priority of *veh_lat* was assigned to 10, which was lower than many other concurrent tasks on P1. The lower priority made the executions of the task *veh_lat* interferred with by all other higher priority tasks, in the worst-case. The task set on P2 was schedulable, but most of the tasks had longer worst-case response times compared to those using RMS in Table 2. The resource utilization for application tasks, timer, and scheduler were 23.15%, 0.56% and 0.0005% for P1, and 0.43%, 0.56% and 0 for P2. As can be seen, the resource consumptions for application tasks remained the same for both processors regardless to the worst-case response times for tasks using the given priorities are longer than those using RMS. This indicates that properly adjusting the priority assignment can improve the system schedulability and timing performance without changing workloads (running some task faster or slower). The timer overheads also remained the same, as both cases used the same resolution. On the other hand, the scheduler consumed less resource in the case of use-specified priorities due to fewer context switches (225 vs 282 in total).

To evaluate the accuracy of the results generated by AIRES tool, we took an approach of comparing the results with a well-known commercial analysis tool, RapidRMA. Ideally, we would like to compare the analysis results with the runtime measurement in the accuracy evaluation. However, due to the

task	period (ms)	execution time (ms)	response time (ms)	utilization	# context switch	priority	processor
V2V.atmio16	21	0.139	2.578	0.007	18	10	P1
V2V.atmioe	2	0.114	0.231	0.057	2	27	P1
V2V.button	30	0.024	2.672	0.001	20	8	P1
V2V.canbrake	8	0.001	0.337	0	5	23	P1
V2V.canfix	20	0.001	0.338	0	6	22	P1
V2V.cani	10000	0	2.799	0	23	3	P1
V2V.canread	7.7	0.104	0.336	0.014	4	24	P1
V2V.cansteer	4	0.001	0.232	0	3	26	P1
V2V.db_slv	1.5	0.037	0.043	0.025	0	29	P1
V2V.DL	1953.9	0.065	2.799	0	22	5	P1
V2V.hmi	200	0.062	2.734	0	21	6	P1
V2V.moblong	21	0.76	2.439	0.036	17	11	P1
V2V.NL	75160	1	3.842	0	25	2	P1
V2V.node1rd	21	0.262	1.448	0.012	13	12	P1
V2V.node1rw	21	0.083	1.186	0.004	12	13	P1
V2V.path101	21	0.047	1.103	0.002	11	14	P1
V2V.pctio10	21	0.107	1.056	0.005	10	15	P1
V2V.radioDriver	29.5	0.07	2.648	0.002	19	9	P1
V2V.regulation	21	0.159	0.943	0.008	9	16	P1
V2V.supervisor	21	0.101	0.784	0.005	8	17	P1
V2V.TL	150108	0	3.842	0	26	1	P1
V2V.veh_iols	21	0.345	0.683	0.016	7	18	P1
V2V.veh_lat	2	0.074	0.117	0.037	1	28	P1

Table 6.8: Analysis results of V2V tasks on P1 using RMS.

availability issues and constraints of the implemented system, we could not run the target code on our platform. The RapidRMA was selected since (i) it is a commercial tool with the functionalities overlapping with AIRES, particularly in timing and schedulability analysis, and (ii) it was also selected by our industrial partner as a baseline tool.

In this experiment, we were interested in how close the results would be between the AIRES tool and RapidRMA. We used the same set of tasks and values to perform the analysis. Table 6.12, 6.13, 6.14, and 6.15 show the analysis results of all tasks for processor P1 and P2 under RMS and user-defined priorities.

The results generated by RapidRMA were very close to the results generated by AIRES. Specifically, the resource utilizations of P1 and P2 were 23.15% and 0.43% respectively for both RMS and user-defined priorities. These were the same as the resource consumed by the application tasks in the AIRES tool. The system overheads, on the other hand, were not considered in RapidRMA. [2]

The response times of tasks of RapidRMA were very close to those of the AIRES tool. The slightly longer response times of the same task of AIRES than those of RapidRMA were caused by the inclusion of OS overheads.

For the user-defined priorities, RapidRMA yielded the same unschedulable decisions for processor P1.

[2]The overheads can be modeled and included in RapidRMA analysis. We did not exercise this since it is not intuitive to include such overheads in RapidRMA.

task	period (ms)	execution time (ms)	response time (ms)	utilization	# context switch	priority	processor
V2V.db_slv	6.4	0.005	0.011	0.001	0	25	P2
V2V.evt300	65.356	0.055	0.121	0.001	4	7	P2
V2V.hi	21	0.007	0.066	0	3	19	P2
V2V.node2rd	21	0.042	0.059	0.002	2	20	P2
V2V.node2wr	21	0.006	0.017	0	1	21	P2
V2V.veh_iomb	5000	0.047	0.168	0	5	4	P2

Table 6.9: Analysis results of V2V tasks on P2 using RMS.

It was the same task *veh_lat* that missed its deadline. The rest of the task set had the similar slightly shorter response times in RapidRMA, which were caused by the inclusion of OS overheads in AIRES analysis.

To complete the analysis of V2V system, the AIRES tool took 5 seconds for both RMS and user-defined priorities. On the other hand, RapidRMA took 73 seconds to complete analysis using RMS and 76 seconds to complete using user-defined priorities. We believe the large computation time difference between AIRES and RapidRMA is caused by the richer functionality and synchronization protocols such as deadline monotonic, cyclic executive, and priority ceiling considered in RapidRMA.

Overall, the experimental results of the ETC application and the V2V application showed that the AIRES tool can be used in any way from an individual tool for certain design analysis to an integrated tool chain through multiple design phases. The AIRES tool provided a unique feature of automatic generation of the runtime model from the software component model. The modeling language enforced the completion of modeling specifications, therefore the performance can be processed with sufficient information. The built-in design process accelerates the design and reduces the errors. The schedulability and timing analysis function provides the better quality of analysis results by including both application workloads and underlying system overheads.

The biggest issue in the current AIRES tool is the seamless integration with other modeling tools into the tool chain. We believe this issue can not be fully resolved without a common modeling language shared by all modeling tools. Although individual translators can be developed for each pair of interactive tools, it would be difficult to maintain these translators as their number will be large if a wide range of candidate tools exist. Consequently, the current AIRES tool can not fully automate the design process. Much labor intensive modeling work has to be done manually, increasing both development time and potentially introducing additional errors. Additionally, the experiments involving runtime measurements are highly desired to make judgement on the quality of the analysis results.

6.2.2 Evaluation with Avionics OEP applications

The experiments of tool chain integration using Avionics were carried out to evaluate and demonstrate the integrability of AIRES tool. In this evaluation, we used software models of Avionics Weapon Systems provided by the Avionics OEP. The OEP had constructed a set of experiment scenarios for such experiments, covering both product scenarios and development scenarios. A product scenario outlines the working aspects and complexity of the system. For example, a basic SP product scenario delt with one or two system threads/transactions running on a single processor with the same rates. A multirate MP scenario was a more complicated system with threads running at multiple rate on multiple processors. A development scenario, on the other hand, defined the development process. In other words, a sequence of steps a design proceeds to construct a system. Avionics OEP defined 11 product scenarios and 10 development scenarios for MoBIES program experiments. We participated in

task	period (ms)	execution time (ms)	response time (ms)	utilization	# context switch	priority	processor
V2V.radioDriver	29.5	0.07	4.926	0.002	16	10	P1
V2V.canfix	20	0.001	0.007	0	0	29	P1
V2V.TL	150108	0	4.926	0	16	10	P1
V2V.DL	1953.9	0.065	4.926	0	16	10	P1
V2V.atmio16	21	0.139	0.591	0.007	6	19	P1
V2V.node1rw	21	0.083	4.926	0.004	16	10	P1
V2V.atmioe	2	0.114	0.616	0.057	6	19	P1
V2V.canread	7.7	0.104	4.926	0.014	16	10	P1
V2V.hmi	200	0.062	4.926	0	16	10	P1
V2V.cani	10000	0	0.083	0	1	25	P1
V2V.NL	75160	1	4.686	0	16	10	P1
V2V.regulation	21	0.159	4.926	0.008	16	10	P1
V2V.canbrake	8	0.001	0.083	0	1	25	P1
V2V.veh_lat	2	0.074	4.926	0.037	16	10	P1
V2V.pctio10	21	0.107	0.616	0.005	6	19	P1
V2V.db_slv	1.5	0.037	0.047	0.025	1	25	P1
V2V.button	30	0.024	0.07	0.001	5	20	P1
V2V.veh_iols	21	0.345	4.926	0.016	16	10	P1
V2V.path101	21	0.047	0.616	0.002	6	19	P1
V2V.cansteer	4	0.001	0.083	0	1	25	P1
V2V.moblong	21	0.76	4.926	0.036	16	10	P1
V2V.supervisor	21	0.101	4.926	0.005	16	10	P1
V2V.node1rd	21	0.262	4.926	0.012	16	10	P1

Table 6.10: Analysis results of V2V tasks on P1 using user-defined priorities.

8 of them that are related to the event and timing modeling and analysis, as shown in Table 6.16.

The size of the model used in each product scenario was different. The basic scenario contained fewer than 50 components. The medium scenario contained around 90 components. And the representative was the largest model, contained around 4000 components.

In the evaluation, the AIRES tool was used as an analysis tool in the tool chain integration. There were two integrated tool chains in the evaluation: one (VU/MI-chain) consisting of Rational Rose translator from Teknowledge, modeling environment from Vanderbilt, the AIRES tool from University of Michigan, and the IIF2AIF translator from Southwest Research Institute; another (H/MI-chain) consisting of the DOME tool from Honeywell and the AIRES tool. Since the tool chain with DOME was performed by the OEP, we only present the results of the experimental results in this report.

The process flow of the experiments is shown in Figure 6.16. The initial models from the OEP were constructed in Rational Rose. These models were converted into GME modeling environment in ESML, and were used as component libraries for the application model construction. The application models were constructed in GME and were exported to AIF format. The AIRES tool took AIF files as input, analyzing the model, and updating the AIF file with the analysis results. The required runtime information were provided by OEP and measured on real target platforms. The raw data was fed to AIF file using SwRI tool.

The tools used in the experiments are listed in Table 6.17.

task	period (ms)	execution time (ms)	response time (ms)	utilization	# context switch	priority	processor
V2V.db_slv	6.4	0.005	0.011	0.001	0	25	P2
V2V.evt300	65.356	0.055	0.215	0.001	1	10	P2
V2V.hi	21	0.007	0.223	0	1	10	P2
V2V.node2rd	21	0.042	0.223	0.002	1	10	P2
V2V.node2wr	21	0.006	0.223	0	1	10	P2
V2V.veh_iomb	5000	0.047	0.223	0	1	10	P2

Table 6.11: Analysis results of V2V tasks on P2 using user-defined priorities.

One important tool performance metric of the Avionics design tool was scalability. We used the computation times to complete an interested analysis as the measure of the scalability. In these experiments, we only recorded the times when the AIF file loading starts until the analysis results are generated. As can be seen from the AIRES for Avionics implementation in Chapter 5, the analyses except the automatic rate and processor allocations have been completed. So the measured times represented the computation times of the analysis. In general, the whole analysis process involved large portions of human interaction, and the times for all aspects of analysis, including translating the model to AIF, annotating execution times from IIF file, performing analyses, reviewing results, and updating the AIF, took between $10 \sim 30$ minutes. Table 6.18 shows the times of analysis in the experiments.

From the measured computation times, the analysis algorithm demonstrated reasonable good scalability. As the number of components increased from $6 \sim 25$ times, the analysis computation times increased only $3 \sim 6$ times for both event analysis and timing analysis algorithms. The execution times of the algorithm increased linearly as the component number increases exponentially. This indicates good scalability of the analysis algorithm. Note that the computation times of the analysis algorithm depended not only on the number of components in the graph, but also the links. This was reflected in the measured times for exp3.3-3.2, which contained fewer components but with a longer computation time.

The evaluation of the automatic rate and component assignment algorithm was also evalauted in a similarly way — by recording the number of components in the system, the number of components allocated, and time to complete the reallocation. We recorded the times to update AIF after the analysis in this experiment since AIF updates were essential to reflect the changes in the modeling environment. Table 6.19 shows the results in this experiment. In this set of experiments, there were only 2 processors. The number given in the parenthesis shows (# of component on P1, # of components on P2).

As can be seen from the results of the automatic assignment experiment, the time for the algorithm was almost constant as the components increased significantly. The computation time also depended on the number of components re-allocated. On contract, the time to update AIF file increased significantly as the number of components in the model increased.

The effectiveness of AIRES tool was evalauted using the number of error detected in the model. Table 6.20 shows the results of errors detected by the AIRES tool. The results showed that we detected errors in the models with designed errors. We also detected some errors that were constructed by designer's mistakes. For example, the representative SP scenario was designed such that it should not have any frameoverrun, but 4 overruns were found after using the IIF data. On the other hand, the frame overrun should exist in the frameoverrun scenario, but no such a case was found with the measured IIF data.

Besides the evaluation of AIRES tool alone, our OEP performed the integrated tool chain and com-

name	global priority	local priority	computation time (us)	relative deadline (ms)	period (ms)	work (ms)
db_slv	1500	1500	37	1500	Det:(1500)	Det:(37)
atmioe	2000	2000	225	2000	Det:(2000)	Det:(114)
veh_lat	2000	2000	225	2000	Det:(2000)	Det:(74)
cansteer	4000	4000	226	4000	Det:(4000)	Det:(1)
canread	7700	7700	330	7700	Det:(7700)	Det:(104)
canbrake	8000	8000	331	8000	Det:(8000)	Det:(1)
canfix	20000	20000	332	20000	Det:(20000)	Det:(1)
atmio16	21000	21000	2560	21000	Det:(21000)	Det:(139)
moblong	21000	21000	2560	21000	Det:(21000)	Det:(760)
nodelrd	21000	21000	2560	21000	Det:(21000)	Det:(262)
nodelwr	21000	21000	2560	21000	Det:(21000)	Det:(83)
path101	21000	21000	2560	21000	Det:(21000)	Det:(47)
pctio10	21000	21000	2560	21000	Det:(21000)	Det:(107)
regulation	21000	21000	2560	21000	Det:(21000)	Det:(159)
supervisor	21000	21000	2560	21000	Det:(21000)	Det:(101)
veh_iols	21000	21000	2560	21000	Det:(21000)	Det:(345)
radiodriver	29500	29500	2630	29500	Det:(29500)	Det:(70)
button	30000	30000	2654	30000	Det:(30000)	Det:(24)
hmi	200000	200000	2716	200000	Det:(200000)	Det:(62)
cani	1000000	1000000	2717	1000000	Det:(1000000)	Det:(1)
DL	1953900	1953900	2782	1953900	Det:(1953900)	Det:(65)
NL	751610000	751610000	3819	751610000	Det:(751610000)	Det:(1000)
TL	150108000	150108000	3820	150108000	Det:(150108000)	Det:(1)

Table 6.12: RapidRMA analysis results of V2V tasks on P1 using RMS.

pared both the VU-chain and the H-chain with the baseline tool chain used in current development. The results showed that both the VU/MI-chain and the H/MI-chain detected some errors that would not be found in the baseline. Particularly, for medium MP scenario, H/MI-chain resulted in 6 times savings in time to find and fix errors. For the representative SP scenario, VU/MI-chain resulted in 11.5 times savings in time to find and fix errors. A $2 \sim 6$ times overall savings in time were observed. Specifically for AIRES tool, the modal and medium MP scenarios were examined. The times for detection of inconsistencies are presented in Table 6.21. As can be seen, the dramatic productivity improvement in integration process was achieved.

name	global priority	local priority	computation time (us)	relative deadline (ms)	period (ms)	work (ms)
db_slv	6400	6400	5	6400	Det:(6400)	Det:(5)
hi	21000	21000	60	21000	Det:(21000)	Det:(7)
node2rd	21000	21000	60	21000	Det:(21000)	Det:(42)
node2wr	21000	21000	60	21000	Det:(21000)	Det:(6)
evt300	65356	65356	115	65356	Det:(65356)	Det:(55)
veh_iomb	5000000	5000000	162	5000000	Det:(5000000)	Det:(47)

Table 6.13: RapidRMA analysis results of V2V tasks on P2 using RMS.

name	global priority	local priority	computation time (us)	relative deadline (ms)	period (ms)	work (ms)
canfix	1	1	1	20000	Det:(20000)	Det:(1)
canbrake	6	6	41	8000	Det:(8000)	Det:(1)
cani	6	6	41	1000000	Det:(1000000)	Det:(1)
cansteer	6	6	41	4000	Det:(4000)	Det:(1)
db_slv	6	6	41	1500	Det:(1500)	Det:(37)
atmio16	8	8	448	21000	Det:(21000)	Det:(139)
atmioe	8	8	448	2000	Det:(2000)	Det:(114)
path101	8	8	448	21000	Det:(21000)	Det:(47)
pctio10	8	8	448	21000	Det:(21000)	Det:(107)
button	10	10	3746	30000	Det:(30000)	Det:(24)
canread	10	10	3746	7700	Det:(7700)	Det:(104)
DL	10	10	3746	1953900	Det:(1953900)	Det:(65)
hmi	10	10	3746	200000	Det:(200000)	Det:(62)
moblong	10	10	3746	21000	Det:(21000)	Det:(760)
NL	10	10	3746	751610000	Det:(751610000)	Det:(1000)
nodelrd	10	10	3746	21000	Det:(21000)	Det:(262)
nodelwr	10	10	3746	21000	Det:(21000)	Det:(83)
radiodriver	10	10	3746	29500	Det:(29500)	Det:(70)
regulation	10	10	3746	21000	Det:(21000)	Det:(159)
supervisor	10	10	3746	21000	Det:(21000)	Det:(101)
TL	10	10	3746	150108000	Det:(150108000)	Det:(1)
veh_iols	10	10	3746	21000	Det:(21000)	Det:(345)
veh_lat	10	10	3595	2000	Det:(2000)	Det:(74)

Table 6.14: RapidRMA analysis results of V2V tasks on P1 using user-defined priorities.

name	global priority	local priority	computation time (us)	relative deadline (ms)	period (ms)	work (ms)
db_slv	6	6	5	6400	Det:(6400)	Det:(5)
evt300	10	10	162	65356	Det:(65356)	Det:(55)
hi	10	10	162	21000	Det:(21000)	Det:(7)
node2rd	10	10	162	21000	Det:(21000)	Det:(42)
node2wr	10	10	162	21000	Det:(21000)	Det:(6)
veh_iomb	10	10	162	5000000	Det:(5000000)	Det:(47)

Table 6.15: RapidRMA analysis results of V2V tasks on P2 using user-defined priorities.

experiment	product scenario	development scenario
exp2.2-1.0	PS 1.10: error scenario SP	DS 2.2: basic SP event analysis
exp2.2-2.1	PS 2.1: representative SP	DS 2.2: basic SP event analysis
exp2-3.4	PS 3.4: medium MP	DS 2: full event analysis
exp2-4.1	PS 3.5: error scenario MP	DS 2: full event analysis
exp3.2-1.9	PS 1.9: frame overrun SP	DS 3.2: SP timing analysis
exp3.2-2.1	PS 2.1: representative SP	DS 3.2: SP timing analysis
exp3.3-3.2	PS 3.2: multirate MP	DS 3.3: basic MP timing analysis
exp3-3.4	PS 3.4: medium MP	DS 3: full timing analysis

Table 6.16: Experiments used in AIRES evalaution.

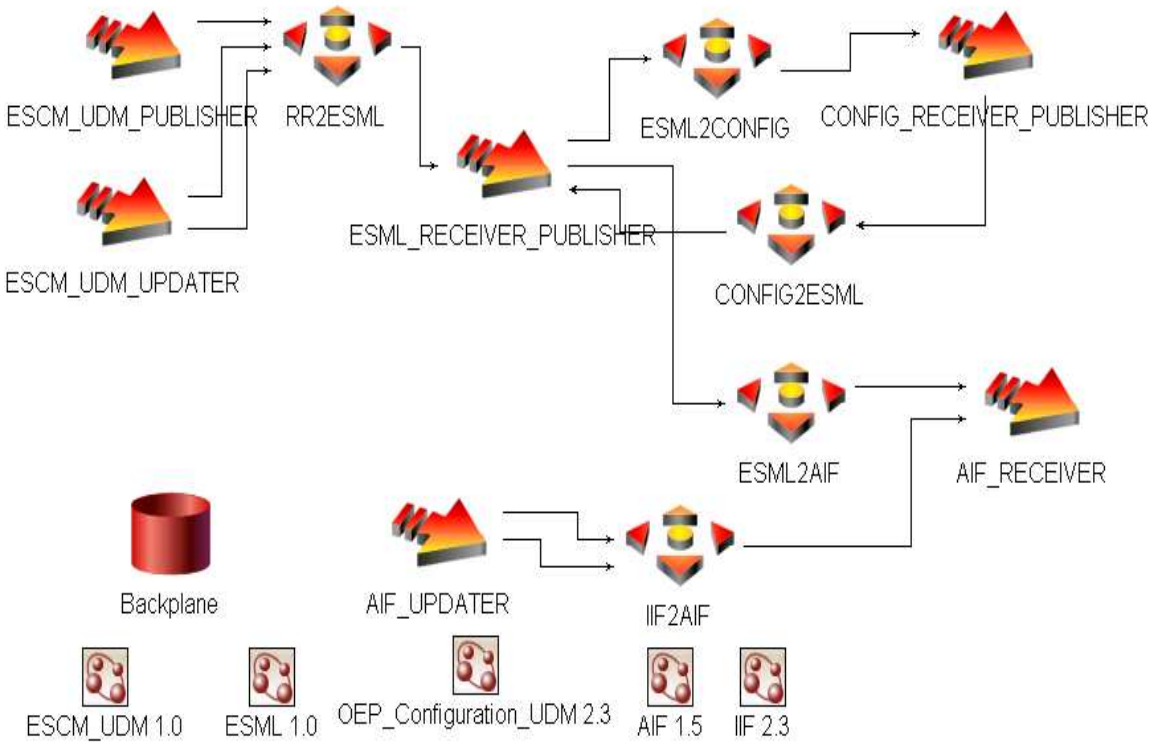


Figure 6.16: Work flow and participant tools in the integration tool chian.

tool	version
GME 3	r3.3.28
ESML	release 04-11-03
AIF	v1.5
AIRES	v2.1
IIF2AIF	v1.3
WOTIF	v1.2.3

Table 6.17: Tools in integration tool chain.

	event analysis				timing analysis			
	exp2.2 -1.10	exp2.2 -2.1	exp2 -3.4	exp2 -4.1	exp3.2 -1.9	exp3.2 -2.1	exp3.3 -3.2	exp3 -3.4
# of component	15	373	90	15	81	373	7	90
time (in sec)	2	12	6	3	3	14	5	6

Table 6.18: Computation time of analysis.

experiment + allocation algorithm	# components original allocation	# of components after reallocation	time for reassignment (sec)	time to update AIF (sec)
MediumMP/first-fit	(48, 42)	(0, 90)	2	4
MediumMP/best-fig	(48, 42)	(48, 42)	1	4
Representative/first-fit	(373,0)	(0, 373)	1	40
Representative/best-fit	(373, 0)	(183,190)	2	28

Table 6.19: Computation times for automatic component assignments.

scenario	event/invocation cycles	no consumer/publisher	frame overrun
ErrorSP	1	1	0
RepresentativeSP	0	3	4
MediumMP	0	7	0
ErrorMP	1	1	0
Frame overrun	0	21	0
MultirateMP	0	1	0

Table 6.20: Errors detected by AIRES in the experiments.

scenario	baseline	VU/MI	H/MI
Modal	18	4	7
Medium	45	4	8

Table 6.21: Time for finding and fix inconsistencies using different tool chains.

Chapter 7

Conclusions and Future Work

In this project, we have developed an approach to integrating non-functional system analyses with model-based embedded system software design. The approach we took has combined the meta-modeling techniques, model-based timing and schedualibity analysis, model transformation and design automation. A model-based analysis toolkit — called the AIRES toolkit — has been implemented. The AIRES toolkit includes the meta-models with non-functional information specifications required by analysis, a set of analysis algorithms for model transformation, functional integration analysis, and timing and schedulability analyses. To ensure the completion of system specification and improve the design/development process, the AIRES toolkit was also implemented with a built-in system analysis process. The process enforces the designer to follow certain steps to perform the analysis, and at each step, a set of minimum system properties must be specified before the analysis can proceed.

To work with the model properties and analysis requirements of different domains, the AIRES tool was tailored to both the avionics and automotive OEPs. Specifically, the AIRES tool for the automotive OEP was implemented using GME as a graphic modeling environment with the designed meta-model and analysis algorithms implemented as interpreters. The tool itself represents an integration tool chain used at different design phases for analysis. We adopted the model translator as a means of integration with models constructed using other tools. In contrast, the AIRES tool for the avionics OEP was implemented as a standalone program using a common file format to exchange models among tools in the tool chain. Both approaches met the objectives of integrating different tools. The approach for the automotive AIRES tool helps perform a richer set of analyses as the modeling information are fully available. However, the algorithm may take a longer time to extract the information required for a certain analysis. The approach for the avionics OEP showed a better performance, but with a cost of modeling information loss.

The evaluations of the AIRES techniques and tool using experiments from both the avionics and automotive domains have demonstrated that the techniques were adequate for such types of embedded applications. It provided useful information to accelerate the design process as well as improve the software quality. The evaluations of individual algorithms using randomly-generated simulation systems showed that the algorithms are of polynormial time with the generated solutions close to the optimal ones. This indicates that the algorithms are suitable for large applications subject to stringent constraints. The evaluations with both avionics and automotive applications showed that the AIRES tool provides useful information on design to satisfy the constraints and contribute to $4 \sim 10$ times savings of development time. It was also shown that more errors were found by the tool than by human designers. The measurement method used with the AIRES tool provided an efficient way of measuring the underlying systems, therefore resulting in more realistic analysis.

We learned several important lessons through this project. First, the design of meta-model has profound impacts on the design and implementation of analysis algorithms. The meta-model defines what analysis-related information is required and in what format. Completing the meta-model and keeping it stable is of the utmost importance for analysis tool implementation and integration. Second, it is possible to automatically refine the initial design to meet the non-functional constraints without human interactions. To achieve this, the quality of service specifications are essential. The quality of the refinment that a tool can achieve depends on the given QoS specifications. The relationships between the specification and the quality of automatically-generated results need further investigation Third, the software architecture, runtime model, and component structure used in the model also have dramatic impacts on the analysis tool implementation and the quality of analysis results. For example, a main factor that the AIRES tool has to be tailored for avionics and automotive applications differently was because the avionics applications use object-oriented models to model components' structure, and use a CORBA middlware-based publish/subscribe for event communications and facet/receptacle for invocations. On the other hand, the components in automotive applications are mainly processing functions, and the software architecture is modeled as data/control flow diagrams. Knowledge of this modeling information and design patterns can help us develop a tool with more efficient algorithms and better integrability.

We believe that AIRES fills a gap in the current software development practice, which relies heavily on time-consuming and expensive testing on the target platform, as it provides insight into non-functional aspects of models at design-level, and helps the engineer make high-level design decisions that have a large impact on the embedded software. It is complementary to tools in a typical IDE (Integrated Development Environment) that works at the code level, such as compilers, debuggers, runtime tracers and automated testers. As the model-based approach is becoming more mainstream, as evidenced by the Model-Driven Architecture initiative [27] and the number of tool vendors in the embedded real-time domain that claim to support it, analysis tools like AIRES that work at the model-level will become more prevalent. The tool is continuously being improved in collaboration with our industry partners, and is available for download at http://kabru.eecs.umich.edu/aires.

We are currently extending the tool in several directions. First, we plan to add more functionalities, especially analysis with real-time network communication. Second, we are collaborating with researchers from other institutes and industries to achieve more seamless integration of AIRES into the end-to-end tool-chain, including feedback of analysis results into the different modeling tools. Third, our timing and schedulability algorithms in the AIRES tool would be conceivable or preferable to interface with mature commercial tools such as TimeWiz [29]. One possible issue with this approach is that commercial tools often have closed and proprietary input/output formats, or does not provide necessary runtime hooks, which may hinder seamless integration within the tool-chain. Last, it will be more valuable to investigate the benefit and improvement our analysis can provide by comparing the design time analysisy results with the runtime measurements of the final executable system. From such a comparison, we can learn the accuracy of the analysis results of the AIRES tool.

Acknowledgements

We would like to thank researchers and engineers at both Avionics and Automotive OEPs for providing us with the models and documentations on the experimental system design, configuration, and execution environments. We would like to thank those at both OEPs for their evaluation of AIRES toolkits and construction of an end-to-end tool chain. We would like to thank researchers at Vanderbilt University for helping us with the ESML language and GME environment, Ben Abbott and Jeremy Price at Southwest Research Institute for their help on IIF2AIF tool, and researchers at Honeywell for DOME tool.

We would like to thank our local industry partners, including ADI International, GM embedded software group at Brighton, GM R & D Department, Ford Research Lab, and ETAS for their valuable feedbacks and discussions.

Last, but not least, we would like to thank the DARPA Program Managers, Janos Sztipanovits and John Bay, for support of this project.

Bibliography

[1] Tarek F. Abdelzaher and Kang G. Shin. Period-based load partitioning and assignment for large real-time applications. *IEEE Transactions on Computers*, 49(1):81–87, 2000.

[2] ARTiSAN Software Tools, Inc., I-Logix,Inc., Rational Software Corp., Telelogic AB, TimeSys Corp., and Tri-Pacific Software Inc. Response to the OMG RFP for schedulability, performance, and time. (revised submission). ftp://ftp.omg.org/pub/docs/ad/01-06-14.pdf, June 2001.

[3] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[4] Iain John Bate. *Scheduling and timing analysis for safety critical real-time systems.* PhD thesis, Department of Computer Science, University of York, November 1998.

[5] Kevin Bradley. *A framework for incorporating real-time analysis into system design processes.* PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1998.

[6] Aaron B. Brown and Margo I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–224, June 1997.

[7] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with *dot*. http://www.reserach.att.com/sw/toos/graphviz, Februray 2002.

[8] Anouck R. Girard, Stephen C. Spry, Paul R. Krets, Susan R. Dickey, Daniel M. Empey, James A. Misener, Pravin P. Varaiya, and Karl J. Hedrick. Vehicle-to-vehicle open experimental platform reference manual. http://robotics.eecs.berkeley.edu/~anouck/v2v_report.pdf, April 2002.

[9] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Fixed priority scheduling periodic tasks with varying execution priority. *Real-Time Systems Symposium*, pages 116–128, December 1991.

[10] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1):13–28, January 1994.

[11] Institute for Software Integrated Systems. The generic modeling environment. http://www.isis.vanderbilt.edu/Projects/gme.

[12] Intel Corporation. Pentium processor family developer's manual, 1997.

[13] Raj Jain. *The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling.* John Wiley & Sons, 1991.

[14] J. Jonsson and K. G. Shin. Deadline assignments in distributed hard real-time systems with relaxed locality constraints. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 432–440, Baltimore, MD, May 27-30 1997.

[15] Kevin A. Kettler, Daniel I. Katcher, and Jay K. Strosnider. A modeling methodology for real-time/multimedia operating systems. In *Proceedings of the Real-Time Technology and Applications Symposium*, pages 15–26, May 1995.

[16] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, (220):671–680, 1983.

[17] P. G. M. Laarhoven and E. H. L. Aarts. *Simulated Annealing: Theory and Applications.* D. Reidel Publishing, 1987.

[18] David J. Lilja. *Measuring computer performance: A practitioner's guide.* Cambridge University Press, 2000.

[19] Larry McVoy and Carl Staelin. *lmbench*: Protable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual TEchnical Conference*, San Diego, CA, January 1996.

[20] Motorola. MPC555 user's manual, revised 15, September 1999.

[21] D.-T. Peng, K. Shin, and T. Abdelzaher. Assignment and scheduling of communicating periodic tasks in distributed real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(12), 1997.

[22] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):412–420, April 1995.

[23] D. Schmidt, D. Levine, and T. Harrison. The design and performance of a real-time corba object event service. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 434–445, 1997.

[24] David Sharp. Reducing avionics software cost through component based product line development. In *Proceedings of the Software Technology Conference*, 1998.

[25] David Sharp. Object-oriented real-time computing for reusable avionics software. In *Proceedings of Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 185–192, 2001.

[26] Jun Sun. *Fixed-priority end-to-end scheduling in distributed real-time systems.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1997.

[27] OMG MDA website. www.omg.org/mda.

[28] OMG UML website. www.omg.org/uml.

[29] TimeSys website. http://www.timesys.com.

[30] Wind River System, Inc. Osekworks 4.0 calls references, 2000.

[31] J. Xu and D. Parnas. On satisfying timing constraints in hard real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–86, January 1993.

[32] Ti-Yen Yen and Wayne Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1125–1136, November 1998.

[33] Lei Zhou. *Real-time performance guarantees in manufacturing systems.* PhD thesis, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan, May 1999.