JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **36,** 66–80 (1996)
ARTICLE NO. 0090

An Overview of RealTimeTalk, a Design Framework for Real-Time Systems

CHRISTER ERIKSSON,* JUKKA MÄKI-TURJA,* KJELL POST,* MIKAEL GUSTAFSSON,* JAN GUSTAFSSON,* KRISTIAN SANDSTRÖM,* AND ELLUS BRORSSON†,[1]

Department of Computer Engineering, Mälardalen University, P.O. Box 883, S-721 73 Västerås, Sweden; and †Department of Computer Engineering, Dalarna University College, P.O. Box 10044, S-781 10 Borlänge, Sweden

RealTimeTalk (RTT) is a design framework for developing distributed real-time applications with both hard and soft requirements. The framework supports design via hierarchical decomposition. We believe that object-orientation is the best way to go about structuring a problem, hence the RTT language is based on Smalltalk with an analysis frontend to infer type information for run-time safety, and to yield more precise estimations of execution times. Unlike most real-time systems, RTT does not force the designer to embed constructs for timing requirements, communication, and synchronization in the code. Rather, such information is specified on a higher level of abstraction using graphical tools. This not only keeps the code "clean" but also simplifies timing analysis and resource allocation. A comparison with other real-time systems concludes the paper. © 1996 Academic Press, Inc.

1. INTRODUCTION

A current trend today is the replacement of mechanical and/or electromechanical control systems, as found in cars and nuclear power plants, with computer-based systems. This move is explained primarily by the reduced production costs and increased functionality and flexibility of software systems. Still, a computer based solution must be at least as dependable as the replaced solution.

Another observable trend is the ever-increasing complexity of computer based systems, a consequence of today's demands on functionality and distribution. In the early days, one computer performed one function, then gradually more and more functions were added to each computer. Today, we see applications requiring a function to be mapped over several computers. Performance and safety issues, as well as geographic reasons, are incentives behind such a distribution of control. These systems are often referred to as real-time systems with the meaning that "the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced" [21].

[1] E-mail: {cen,jma,kpt,mgn,jgn,ksm}@mdh.se; jbr@t.hfb.se.

In the real-time research community, one often distinguishes between *hard* and *soft* temporal requirements. If a deadline is violated in a hard real-time computation the logical result is useless. In a soft real-time computation, the logical result could have a meaning even though a deadline is not met. Many real-time applications of today have both hard and soft temporal requirements. For example, in the computer system of a car, a hard requirement involves controlling the brakes, whereas the automatic climate control is an example of a soft requirement.

Real-time applications often have two types of requirements on control: *periodic* and *aperiodic*. Periodic control means that an activity is computed repeatedly with a predefined period time, e.g., implementing a cruise control for a car leads to a periodic activity. Aperiodic control means that the application generates events that the computer system must respond to within a certain time, e.g., the brake system in a car. Periodic and aperiodic activities can be either soft or hard.

An important aspect in the design of real-time systems is the integration of methods, architectures, and tools to avoid inconsistency between design and implementation. It is not unusual to find people using a particular method in the design phase of some new system, then working with another tool for the implementation, violating specifications set during the design phase. A classical example of this dilemma is structured analysis and design which aids the specification of the system without considering the runtime environment. With the proper integration of design specification and implementation tools, specifications made in the design phase could be enforced throughout the entire life-cycle of the product.

This paper describes RealTimeTalk (RTT), a framework for the development of applications with both hard and soft real-time requirements. The organization of the paper is as follows. Section 2 highlights the features of RTT; Section 3 presents the programming-in-the-large aspects of RTT, followed by a presentation of the RTT language in Section 4. Section 5 describes the run-time environment; Section 6 describes the prototype system; Section 7 con-

0743-7315/96 $18.00
Copyright © 1996 by Academic Press, Inc.
All rights of reproduction in any form reserved.

tains a short description of related work. We conclude the paper in Section 8.

2. INTRODUCTION TO RTT

The main goal of RTT is to simplify the design and implementation of predictable real-time systems. We strongly believe that object-oriented methods are of vital importance in this process. Object-orientation gives advantages such as reusability, rapid prototyping, incremental development, modularity, extensibility, and promotes the use of frameworks. However, notions for synchronization and distribution are often weak.

A number of desirable properties for the development of real-time systems can be identified (cf. [6]). For instance, frameworks which support hierarchical decomposition fosters good design and helps the designer to cope with complexity. Component based design promotes reusability and safety. The transformation between analysis, design, and implementation should be reversible in order to maintain the consistency between the different models when the system is maintained and extended. Within the framework, it should be possible to both specify and verify the functional and temporal aspects of the system. Ideally, a common description language for different levels of abstraction should be used. This language should support incremental development, prototyping and simulation, error-handling mechanisms, and the possibility of encapsulating low level languages, for reasons of efficiency and code re-use.

The *design framework* for RTT is divided into parts for programming in the large and programming in the small. Programming in the large corresponds to application design on a higher level and contains tools to map a design to a resource structure, whereas programming in the small concerns implementation of classes.

The syntax of the RTT programming language is based on Smalltalk [8], a true object-oriented language with simple syntax and semantics. Recently, Smalltalk has begun to regain some of the ground it lost after C++'s appearance. We believe this trend is mainly due to Smalltalk's simplistic nature and its support for rapid prototyping, making it very easy to focus on the problem and to quickly build high quality applications without having to worry about, e.g., memory management—a nontrivial problem in most C++ applications.

However, the nondeterministic nature of garbage collection and method invocation, together with the weak type system makes Smalltalk less suited for real-time applications. The language also has weak support for communication and synchronization in a distributed environment. In our adaptation of Smalltalk, we have modified these drawbacks to make RTT type safe, predictable, and usable in a distributed environment.

The RTT design framework also separates the specification of hard and soft real-time into two parts. It also includes the interface between these two parts. The software model for the design of the hard real-time part of an application is based on a set of design objects where hierarchical decomposition is a key concept. The soft real-time part is currently open for any reactive model that conforms to the interface of the hard real-time part and is adaptable to the RTT run-time system.

In the hard real-time part, we have separated the functional and temporal behavior of the system. The functional requirements are specified with the RTT design objects and are checked by prototyping and testing. The temporal constraints are also specified within the RTT design objects and is statically verified by the RTT pre-run-time scheduler. The maximum calculated execution time (MAXTC) is calculated for each schedulable entity (task) by the RTT compiler. The timing information is then fed to the scheduler which tries to find a feasible schedule for the system. If such a schedule is found the temporal behavior will be guaranteed during run-time.

The run-time environment consists of a set of nodes which are connected by a predictable broadcast bus. The RTT software platform runs on each node and is divided into two parts: an operating system and a communication system. The software platform supports execution of time-triggered hard real-time tasks and event-triggered soft real-time tasks. Furthermore, the communication system supports both hard and soft real-time messages.

3. PROGRAMMING IN THE LARGE

3.1. Introduction

When designing object-oriented systems today, one often ends up with a monolithic structure where an ocean of objects collaborate to achieve a desired function. Needless to say, maintenance of such nonhierarchical implementations are difficult at best and requires understanding of the system on all levels. It also leads to problems for the customer, who has to verify the functionality of the actual implementation against the requirements specification. Of course, many systems are described on different levels of abstraction during the design, but these abstractions are seldom explicit in the implementation. Another issue is how to handle synchronization and communication in an object-oriented system. Most programming models integrate synchronization and communication with the functionality. For example, when using a real-time operating system, tasks typically have synchronization calls interleaved with the rest of the code, making it difficult to verify and maintain the code.

Furthermore, when implementing time constraint services with strict deadlines in a conventional system, time constraints are often mapped to period times or priorities of individual tasks. This approach is often usable when constructing simple single node systems, but when implementing time constraint functions which include several tasks (and where the tasks might execute on different nodes) these solutions are not adequate. To implement these functions, the time constraints must be specified for

the complete function in an explicit way to make the system predictable.

Implementing a time constrained service can be done either by the *time-triggered* or *event-triggered* approach. The time-triggered execution model is defined as follows: the system observes the state of the environment at specific points in time. Thereafter the system decides, based on an analysis of the state, which actions must be taken. Afterwards, new values are emitted to the environment at a predefined point in time. By event-triggered, we mean that an event occurrence propagates into the control system at an arbitrary point in time. For example, an interreupt normally initiates an action, such as resuming a task waiting for the event.

When implementing hard real-time functions it is very important to be able to verify the function easily. With the time-triggered approach this is more straightforward, because the number of states to test is much smaller than for the event-triggered approach [15]. The reason for this is that the events in the environment can only propagate into the system at predefined points in time, or more specific, in predefined time intervals. In the event-triggered model, events can propagate into the system at arbitrary points in time and thus the temporal behavior will be more difficult to verify compared to the time-triggered approach. As a consequence, the time-triggered approach is superior to the event-triggered approach when implementing control loops and monitoring functions. On the other hand, when implementing functions that are inherently event-triggered, the transformation to a time-triggered model is often nontrivial and the transformed solution does not reflect the structure of the function. Thus, when implementing such functions the event-triggered approach is more suited. Furthermore there are powerful, commercially available tools for modelling event-triggered client-server applications, for instance user interfaces. We conclude that there is a need for both of these approaches, because one could not choose an approach without considering the controlled application.

RTT is well suited for time-triggered hard real-time applications, and periodic or event-triggered soft real-time applications.

3.2. Application Example

The following example, in which we imagine a railroad network supervised by a distributed computer system serves to illustrate various aspects of the RTT software model.

The railroad network is divided into segments (Fig. 1). Segments can be connected in either end to form loops and junctions. Each segment consist of a straight sequence of zones, each zone corresponding to a fixed amount of rail. Trains in a segment are managed by a segment computer. Trains are considered dumb in that respect that the segment computer dictates their speed. Segment computers need to negotiate on in- and outgoing trains and are therefore linked to each other. A central computer (henceforth referred to as the traffic area controller, TAC) collects information from each segment computer to construct an overall plan. This plan is produced continuously to make efficient use of the railroad network. The segment computers attempts to follow this plan, but will of course deviate from it, should there be a chance of collision. It should be stressed that segment computers are in no way dependent on the TAC for safety issues.

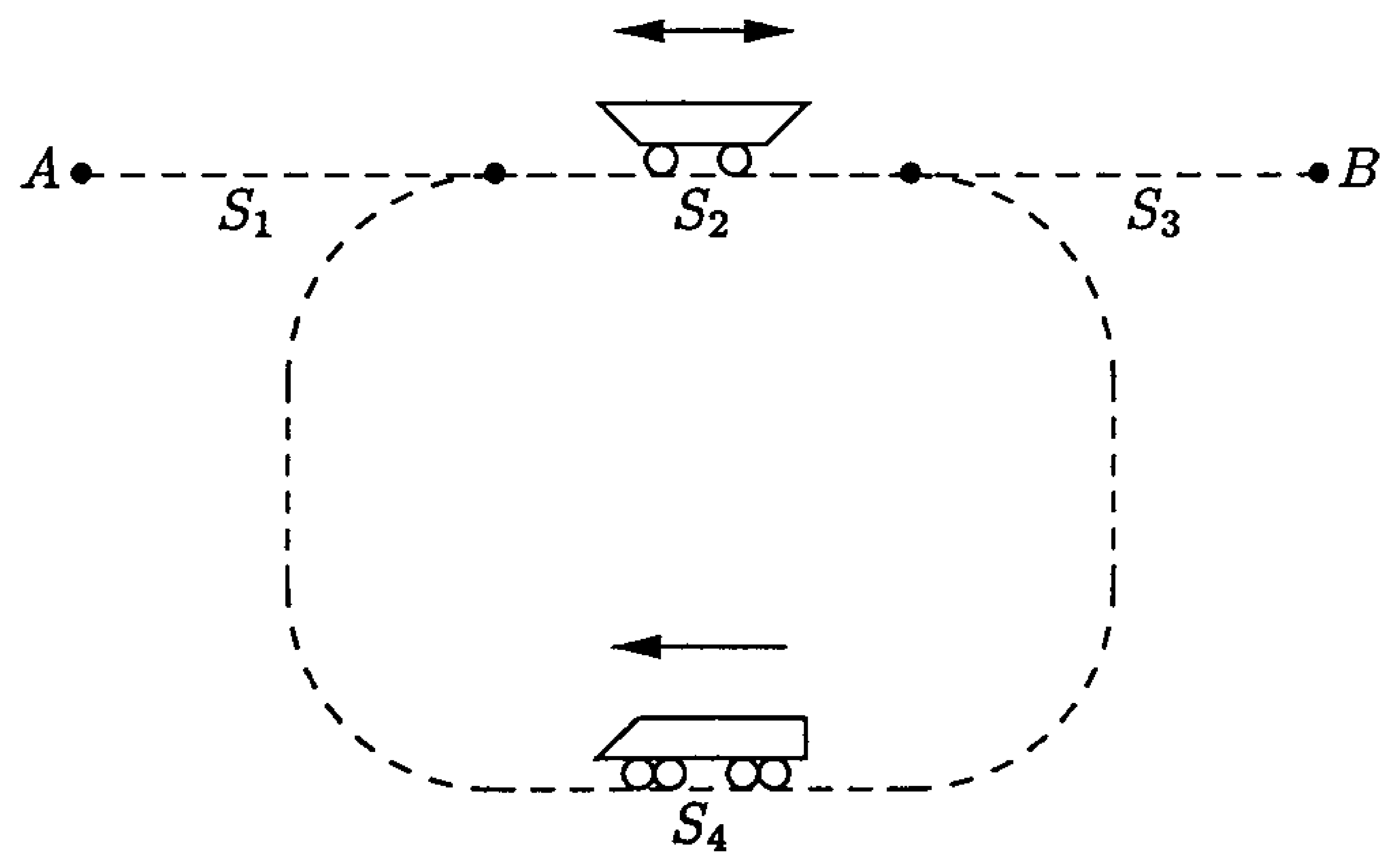


FIG. 1. Railroad network. Each S_i denotes a segment; dashes correspond to zones.

A plan is based on a "snapshot" of the system and describes desired speed settings for each train until the next plan is constructed. After the plan is formed (which may take some time), a new snapshot is taken to verify that the plan is still consistent. If so, the plan is downloaded to the segment computers, otherwise it is discarded. Should the central computer face a deadlock, the system stops and an operator have to resolve the situation.

This example will be used in the remainder of this paper to illustrate various parts of the RTT framework.

3.3. The Hard Real-Time Model

A key concept in the design of hard real-time applications is the use of different levels of abstractions. In RTT, design is supported by hierarchical decomposition and the use of a predefined set of design objects. These design objects are *modes*, *mode transitions*, *use-cases*, and *tasks*. Figure 2 illustrates a generic application structure.

3.3.1. Modes and Mode Transitions. When developing a system one often realizes that the modelled system has several distinct states, or *modes*. For a locomotive, these modes could be notOperating, manuallyOperating, and autopilot. The fact that there exist distinct modes does not mean that the functionality of these modes are disjoint. For instance, it is reasonable to assume that the modes manualOperation and autopilot have some common functionality. When the different modes of a system have been identified one has to model the transitions between them. In the train example, two of the mode transitions could be ManualToStopped and AutoToManual.

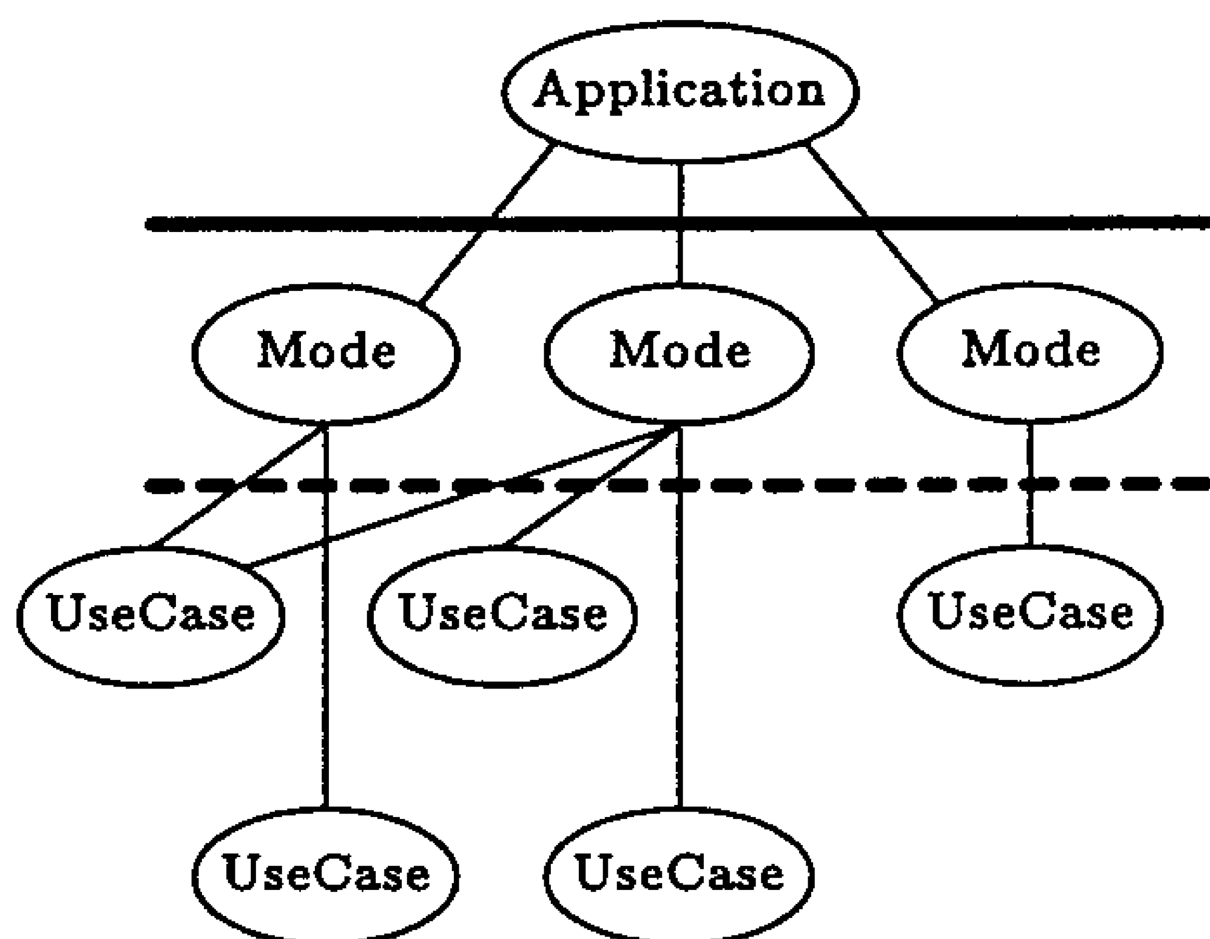


FIG. 2. The hierarchical structure of a generic application.

A mode defines the activities that must take part during a particular state of the application. A mode transition corresponds to actions taken when we have a change of mode in the system. Thus a mode is mapped to a continuous schedule whereas a mode transition is mapped to a one-shot schedule.

The problem with implementing modes and mode transitions is that most systems does not have an explicit notation for modes. Instead, modes and mode transitions are usually embedded in the code. If the model supports distinct modes one can easily see which functions that run in each mode and thereby allocate resources in an efficient way. Another advantage is that each mode could be designed separately. In the RTT model, a precise notation and semantics for modes and mode transitions are supported by the design objects Mode and ModeTransition. The transitions between the different modes are described as a high level state machine. For instance, in Fig. 3, we see such a graph for the train application.

In RTT each activity in a mode and mode transition is modelled by a use-case.

3.3.2. Use-Cases. A use-case can be seen as an engine that controls a number of cooperating objects to perform a certain task. A use-case models an activity as a periodic function. An aperiodic activity is translated to a periodic activity, for example, with the theory provided by Mok [18].

When implementing a use-case that consists of a several collaborating computational entities, it is important to have an explicit notation for both synchronization and communication, as well as being able to describe both sequential and parallel executions. With synchronization we mean the order in which the different entities should be executed. Furthermore, it should also be possible to describe that some resource is shared and that the access of this resource is protected. In many systems these mechanisms are, as mentioned before, embedded in the implementation which make them hard to understand and maintain.

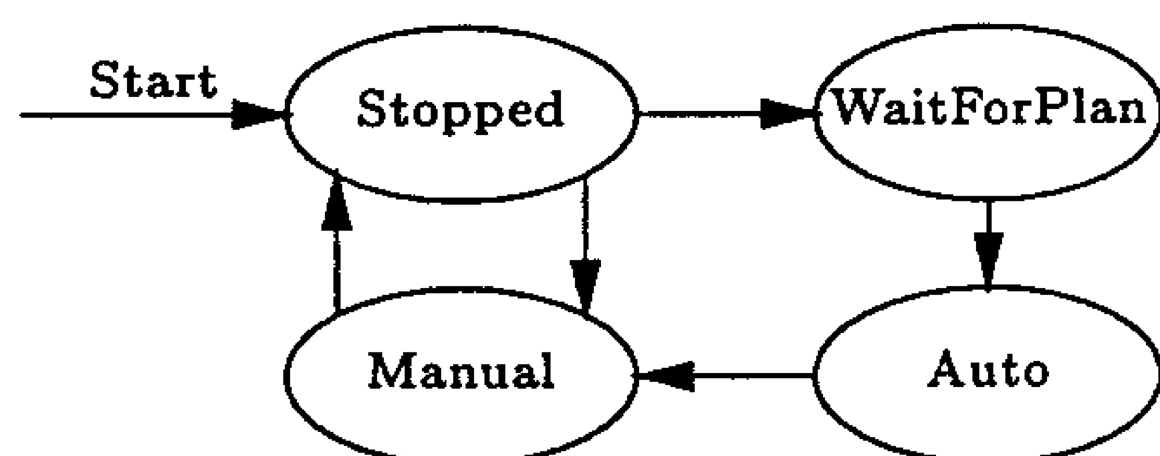


FIG. 3. A high-level state machine for the train example.

With communication we mean the information that is exchanged between the different entities. This information should also be explicit for the same reasons as for synchronization. RTT models both *synchronization* and *communication* with a precise syntax and semantics, at a high level of abstraction.

A use-case is defined by a *precedence graph*, an *interaction graph*, and the *period time* of the computation. A precedence graph is a directed acyclic graph and defines the execution order between the entities. The interaction graph specifies the communication between the entities and the use of shared resources. The period time specifies the time between two consecutive activations of the entities specified in the precedence graph. These entities are called tasks.

3.3.3. Tasks. A task encapsulates an object and provides the thread of control and communication for the object. As just mentioned, the model does not mandate synchronization and communication constructions in the code. Rather, we have equipped the tasks with in- and out-ports to decouple the code from communication details. The synchronization is decoupled from the code by explicitly specifying it in a precedence graph. Let us first describe the operation of a task, then its temporal attributes.

Assume an object S with a method `m:` and an object C who wants to send the message `m: arg` to S (Fig. 4). C could potentially reside on a different node in the system. As seen in Fig. 4, the object S and its method are encapsulated by a task. The role of the task is to (1) map the in-port of the task to the arguments of the method; (2) invoke the method `m` in the object; and (3) map the result of the method's execution to the out-ports.

The temporal attributes of the task are *release time*, *deadline*, *MAXTC* (maximum calculated execution time), and *MINTC* (minimum calculated execution time). These temporal attributes can be divided into two groups. The

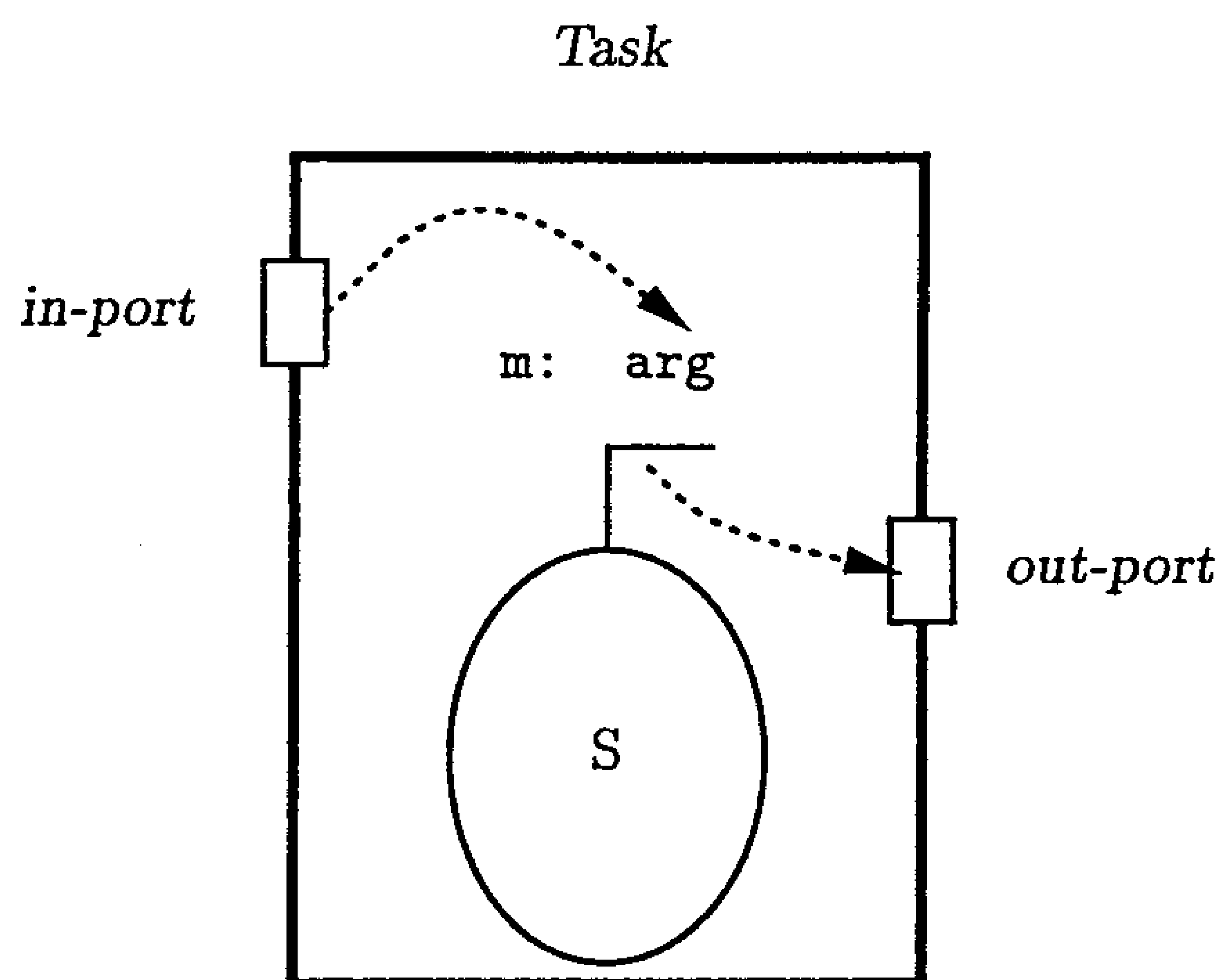


FIG. 4. The structure of the task that encapsulates the object S.

first group, involving MAXTC and MINTC are static, meaning that these parameters do not depend on the use-case the task is part of. Release time and deadline, on the other hand, are context dependent; i.e., they depend on the temporal requirements of the use-case in which they are defined. The release time and deadline are set relative to the period time of the specific use-case. The following relation defines constraints on the release time and deadline:

$$(\text{deadline} \leq \text{period time}) \wedge (\text{release time} \geq 0) \wedge (\text{deadline} \geq \text{release time} + \text{MAXTC}).$$

A complete specification of a task include the object that it encapsulates, the activation method of the encapsulated object, the node the task will execute on, the in- and out-ports, and the four temporal attributes just mentioned. In RTT, a port is either a primitive object—such as a boolean, integer, character, float or string—or a compound object, having primitive objects as instance variables.

3.3.4. Precedence Graphs and Interaction Graphs. For some activities, it is very important that input and output with the environment is synchronized. For example, in the train example it is imperative that each snapshot of the trains' position is taken within a certain time interval by each segment computer, or otherwise an inconsistent view of the state may result. It is also desirable to be able to describe the execution order of the tasks.

In order to specify synchronization between tasks on a higher level of abstraction, we have introduced precedence graphs. A precedence graph is a directed acyclic graph which specifies the precedence relationship between tasks. For example, if task *A* precedes task *B* in the graph, task *A* must terminate its execution before task *B* can start.

To specify communication between tasks, the *interaction graph* is introduced. The interaction graph is a binary relationship specifying pairs of tasks that can communicate with each other, i.e., where the producer's out-port is connected to the consumer's in-port. The interaction graph also specifies the use of shared resources.

These two graphs are often merged into one graph. For example, in the MARS system [16] the precedence relationship also includes data transfer. In our opinion, this requirement is too strong and makes it difficult to describe things like feedback loops. The reason behind our separation of synchronization and communication is that these two issues really are orthogonal concepts; i.e., communication may or may not be synchronized. Figure 5 shows a precedence graph for the use-case in the train example that takes snapshots of the state of the environment. S_n is the task that encapsulates the segment object at segment computer n. The idea is that each segment computer is supposed to take a snapshot of the trains' position at roughly the same time. When the snapshot has been taken the information is sent to the TAC task which encapsulates the traffic area controller object.

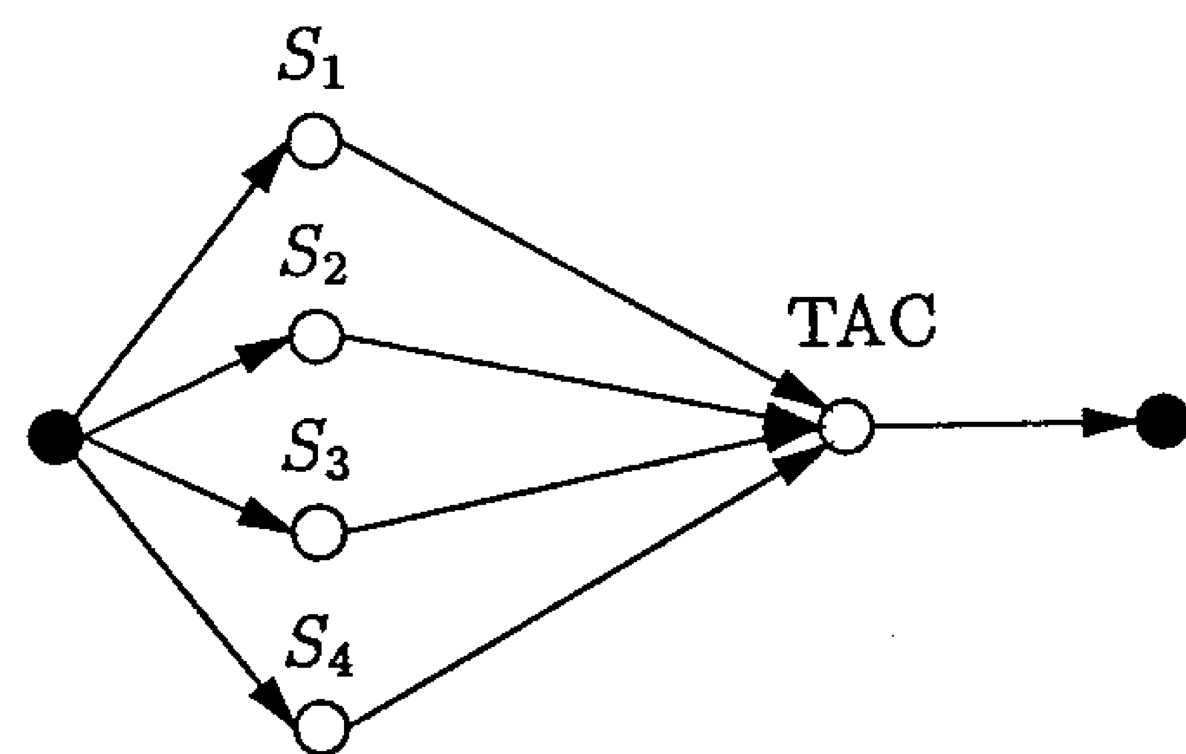


FIG. 5. The precedence graph for the use-case snapshot.

The corresponding interaction graph is shown in Fig. 6. In this example, the communication is synchronized; i.e., each precedence relation has an associated interaction relation.

The precedence graph, the period time of the use-case, and the tasks' temporal attributes are related, as seen in the following example.

EXAMPLE 3.1. The time between two consecutive activations of the tasks in the precedence graph is specified by the period time of the use-case. Assume that a period time of 500 ms has been derived from the requirements specification to obtain the needed observation frequency. The precision on each observation requires that the observation of each segment object is done within 100 ms. The precision requirement can be specified by the release time and deadline attribute of each task. In this case, the release time could be specified as 0 ms for the segment tasks (*S*1 ... *S*4). The deadlines for each segment task then has to be specified to 100 ms to fulfill the precision requirement.

In many applications, sensors are read with a much higher frequency than the control loop executes to enable signal processing [25]. Therefore it must be possible to specify the communication between a task in the control loop's use-case and the producing signal's processing task. This is specified in the same way as for tasks communicating within one use-case, i.e., by connecting the producing task's out-port to the consumer's in-port. There is no difference between these two cases from the modelling point of view but the translation to a resource structure will be different. This translation will be described in Section 6.5.1.

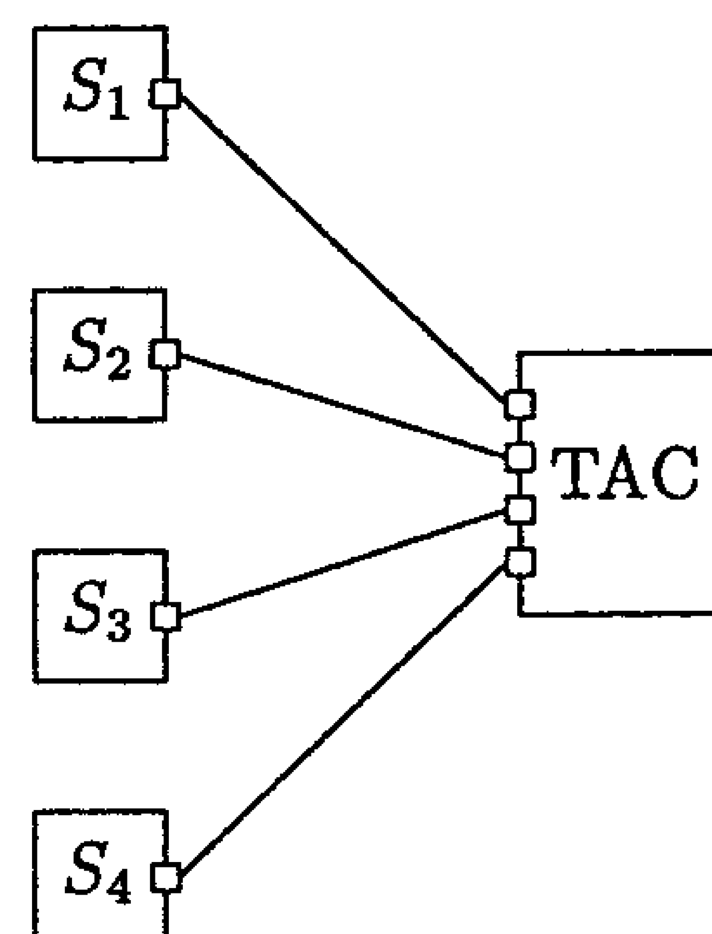


FIG. 6. The interaction graph for the use-case snapshot.

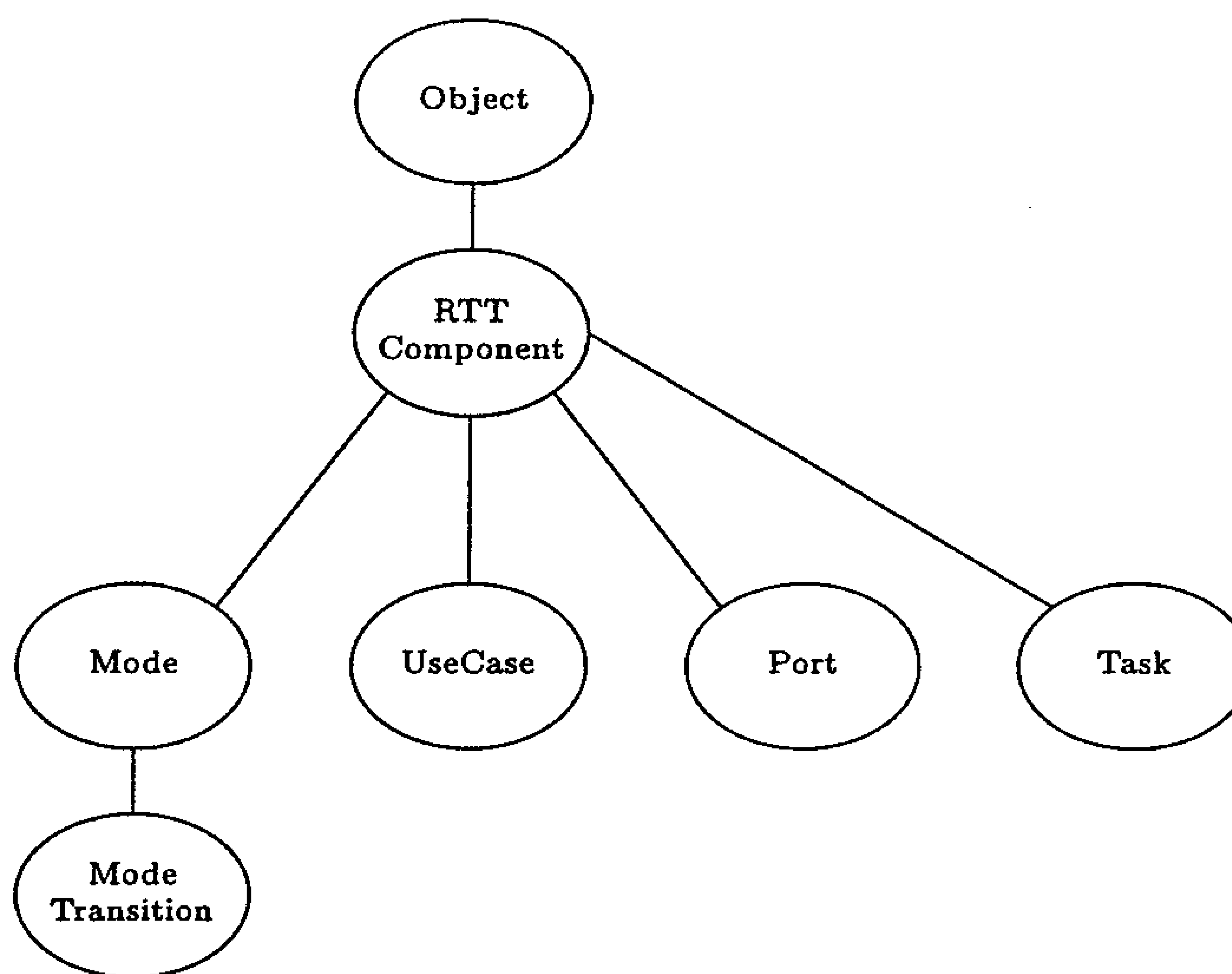


FIG. 7. Class hierarchy.

3.3.5. Supporting Class Hierarchy. The RTT class hierarchy supports objects such as modes, mode transitions, use-cases, and ports. An application engineer uses these classes when developing his system. Furthermore these objects are mapped to the resource structure to reflect the design; i.e., one can easily follow the high-level specification all the way down to implementation. An example of such a hierarchy can be seen in Fig. 7.

3.4. The Soft Real-Time Model

The reason for introducing a soft real-time model is that most applications of a reasonable size have both hard and soft functionality. The question is: why not implement the soft functions as hard? There are several reasons:

- If resources for a soft function is allocated pre-runtime, the period time and the maximum execution time must be considered. If a short response time is required for an event, the period time must be relatively short. This, however, gives a poor utilization if the event occurs seldom. On the other hand, if one would like to increase the utilization the period time must be increased but then the responsiveness will decrease. If the function instead is implemented in the soft real-time part the assumption is that the system in the average case will have an acceptable response time. However, one should keep in mind that no guarantee is provided.
- Sometimes it is very difficult to model a inherently event-triggered function in a time-triggered model.
- If the hard and soft functions are implemented in separate parts, the effort that must be spent on verification and validation of timing requirements will decrease. Many hard real-time functions are also safety critical and if the safety critical functions are separated from the rest of the system, the verification and validation of the safety requirements will be simpler [17].

Another reason for having both hard and soft capabilities is that this can be used to give the hard part an increased flexibility. For instance, making on-line modifications to a system, e.g., installing a new schedule and modified code. When the soft part has loaded the alterations to the system, the hard part would be informed and a change to the new schedule could be made at an appropriate time. This functionality would be very expensive to implement with just the hard part because, as mentioned earlier, the resources to accomplish this have to be pre-allocated and this functionality will be used seldom.

In the case, where the hard and soft part has to cooperate, i.e., exchange information and/or control, there must be clear definitions how to perform this cooperation without jeopardizing the temporal requirements for the hard real-time part and the consistency of data for both the hard and the soft parts. The information that is passed between the soft and hard parts is, as defined earlier, residing in objects. The responsibility of the interface could either be delegated to each object that holds the information, or to special interface objects, acting as a fire wall between the two parts. The benefit with mapping the interface direct to objects is that the object-oriented paradigm is not suppressed. The disadvantage of this approach is that failures in the soft part more easily could intrude on the hard real-time part. If special interface objects are used instead, the object-oriented approach could not be fully utilized because shared information has to be spread out on several objects. The advantage is that the separation of the two parts are distinct.

Independent of which model is adopted, data consistency must be maintained in both directions. How to achieve this has been investigated by, for example, Thijssen *et al.* [19].

Depending on the application to develop both approaches could be considered. If there are high demands

on safety the fire wall approach should be used. On the other hand if the application does not have high demands on safety but instead have a high structural complexity the interface on the object level is preferred to minimize the suppression of the object-oriented paradigm. In RTT, both these approaches are available so the developer could choose the appropriate one.

4. PROGRAMMING IN THE SMALL—THE RTT LANGUAGE

4.1. Introduction

The RTT language is primarily targeted toward the code for the hard real-time part and we will discuss the language from this point of view. Naturally one could use RTT also for the soft real-time part, without the restrictions discussed in this section.

A feature brought over from the Smalltalk community is the notion of frameworks, i.e., predefined class hierarchies with support for different application areas. This will speed up the development of applications and promote high quality code and reuse.

An underlying design philosophy in RTT is to let the designer produce a prototype within a Smalltalk development environment, and with limited concern for temporal aspects. The prototype can be implemented and modified quickly. This prototyping environment helps the programmer to focus on a reasonable functional solution to the problem at hand. Later, when the designer is satisfied, the code is fed to the RTT compiler which provides information about execution times for each task to the RTT scheduler, which in turn tries to find a feasible schedule.

4.2. Syntax and Semantics

Although the syntax of RTT is the same as of Smalltalk, the semantics of some programming constructs have been changed to make applications predictable:

1. Recursion is not allowed because of the problem of determining the recursive depth of a data dependent recursion [9].

2. Loops have to be bounded. Constructs in Smalltalk like

```
[...] whileTrue: [...].
```

with no upper bound on the number of iterations have in RTT been replaced with

```
[...] whileTrue: [...] maxIterations: m.
```

These and other similar constructs are defined in RTT and may not be changed by the application programmer. For a detailed description, please see [9].

3. Dynamic creation or changes of classes and methods are not allowed at run-time. This is necessary to guarantee time determinism of applications.

4. Data structures have to be limited in size; e.g., linked lists of indefinite length are ruled out.

4.3. The Effect of Polymorphism

RTT shares Smalltalk's dynamic typing; i.e., types are associated with values rather than variables. Dynamic typing is generally considered more flexible than static typing and also relives the programmer from having to declare the types of variables; this is a benefit, especially during the prototyping phase.

However, because of the lack of typing information at compile time, the compiler must make pessimistic assumptions regarding which method will be invoked for a certain message sent at run-time. This leads to an over-estimation in the calculation of the MAXTCs. A more serious problem is the risk of getting "message not understood" at run-time; i.e., the receiving object does not implement a method for the message. This is of course disastrous in a hard real-time system.

One ad hoc way of dealing with this problem is to systematically rename methods so that it will always be clear by looking at the program text which method will be invoked for a particular message. This, however, is against the very idea of polymorphism. Another alternative is to obtain information about which class the receiver of the message is an instance of. This can be arranged by letting the programmer provide type declarations. However, we would like to be able to use as many Smalltalk programs as possible "as they are," keeping manual conversion chores to a minimum. We also believe that type declarations are a burden to the programmer, especially during the prototyping phase.

Instead, we are currently in the process of designing a type inference system for RTT [10]. The purpose of this system is to annotate each variable occurrence in the program with a type. We define a type as a set of class names $\{C_1, ..., C_k\}$ that represents the classes that a variable occurrence can be an instance of at run-time.

With type inference, RTT can be used "typeless" in the prototyping phase. Later, when the product is ready to run, type inference is used to detect type errors and optimize the execution (Fig. 8).

With type information at hand, the compiler can

• *Reduce over-estimations of MAXTC calculations.* When a receiver is known to be an instance of a set of classes, the MAXTC calculation is limited to these classes.

• *Guarantee statically type-safe programs.* Once types are inferred, the compiler can verify statically that all messages will be understood at run-time and that arguments

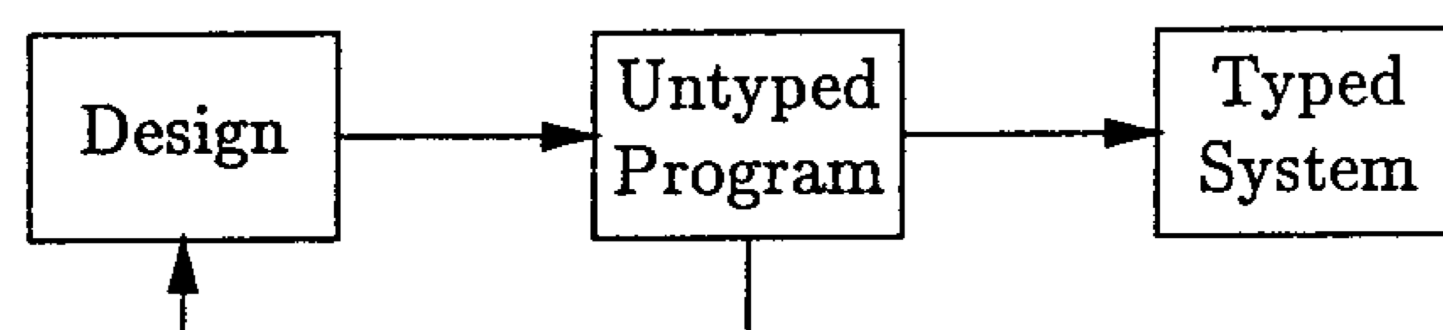


FIG. 8. Type inference in RTT program development.

on safety the fire wall approach should be used. On the other hand if the application does not have high demands on safety but instead have a high structural complexity the interface on the object level is preferred to minimize the suppression of the object-oriented paradigm. In RTT, both these approaches are available so the developer could choose the appropriate one.

4. PROGRAMMING IN THE SMALL—THE RTT LANGUAGE

4.1. Introduction

The RTT language is primarily targeted toward the code for the hard real-time part and we will discuss the language from this point of view. Naturally one could use RTT also for the soft real-time part, without the restrictions discussed in this section.

A feature brought over from the Smalltalk community is the notion of frameworks, i.e., predefined class hierarchies with support for different application areas. This will speed up the development of applications and promote high quality code and reuse.

An underlying design philosophy in RTT is to let the designer produce a prototype within a Smalltalk development environment, and with limited concern for temporal aspects. The prototype can be implemented and modified quickly. This prototyping environment helps the programmer to focus on a reasonable functional solution to the problem at hand. Later, when the designer is satisfied, the code is fed to the RTT compiler which provides information about execution times for each task to the RTT scheduler, which in turn tries to find a feasible schedule.

4.2. Syntax and Semantics

Although the syntax of RTT is the same as of Smalltalk, the semantics of some programming constructs have been changed to make applications predictable:

1. Recursion is not allowed because of the problem of determining the recursive depth of a data dependent recursion [9].
2. Loops have to be bounded. Constructs in Smalltalk like

```
[...] whileTrue: [...].
```

with no upper bound on the number of iterations have in RTT been replaced with

```
[...] whileTrue: [...] maxIterations: m.
```

These and other similar constructs are defined in RTT and may not be changed by the application programmer. For a detailed description, please see [9].

3. Dynamic creation or changes of classes and methods are not allowed at run-time. This is necessary to guarantee time determinism of applications.
4. Data structures have to be limited in size; e.g., linked lists of indefinite length are ruled out.

4.3. The Effect of Polymorphism

RTT shares Smalltalk's dynamic typing; i.e., types are associated with values rather than variables. Dynamic typing is generally considered more flexible than static typing and also relives the programmer from having to declare the types of variables; this is a benefit, especially during the prototyping phase.

However, because of the lack of typing information at compile time, the compiler must make pessimistic assumptions regarding which method will be invoked for a certain message sent at run-time. This leads to an over-estimation in the calculation of the MAXTCs. A more serious problem is the risk of getting "message not understood" at run-time; i.e., the receiving object does not implement a method for the message. This is of course disastrous in a hard real-time system.

One ad hoc way of dealing with this problem is to systematically rename methods so that it will always be clear by looking at the program text which method will be invoked for a particular message. This, however, is against the very idea of polymorphism. Another alternative is to obtain information about which class the receiver of the message is an instance of. This can be arranged by letting the programmer provide type declarations. However, we would like to be able to use as many Smalltalk programs as possible "as they are," keeping manual conversion chores to a minimum. We also believe that type declarations are a burden to the programmer, especially during the prototyping phase.

Instead, we are currently in the process of designing a type inference system for RTT [10]. The purpose of this system is to annotate each variable occurrence in the program with a type. We define a type as a set of class names $\{C_1, ..., C_k\}$ that represents the classes that a variable occurrence can be an instance of at run-time.

With type inference, RTT can be used "typeless" in the prototyping phase. Later, when the product is ready to run, type inference is used to detect type errors and optimize the execution (Fig. 8).

With type information at hand, the compiler can

- *Reduce over-estimations of MAXTC calculations.* When a receiver is known to be an instance of a set of classes, the MAXTC calculation is limited to these classes.
- *Guarantee statically type-safe programs.* Once types are inferred, the compiler can verify statically that all messages will be understood at run-time and that arguments

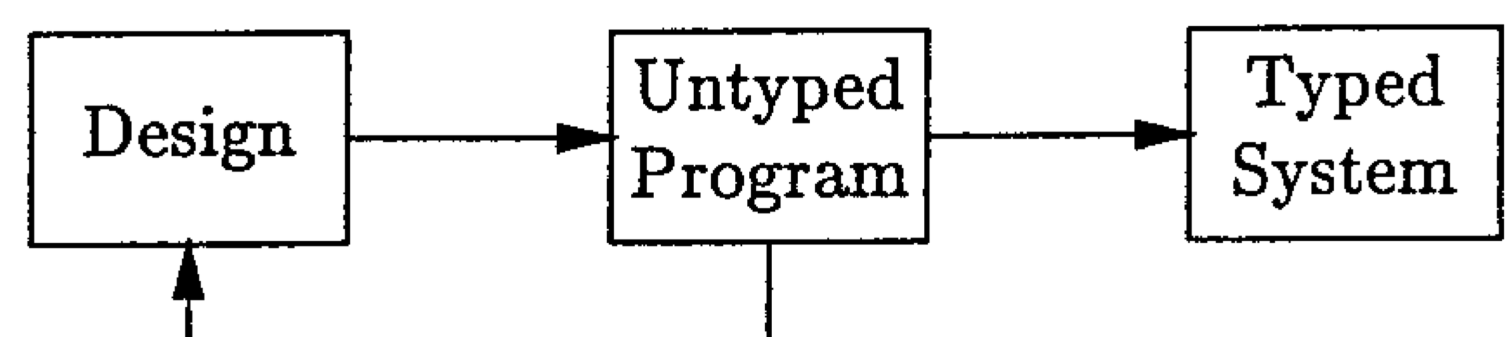


FIG. 8. Type inference in RTT program development.

to system builtins have the proper type. Of course, the compiler may be overly pessimistic and reject a program that would avoid an offending construct at execution.

- *Eliminate run-time type checking.* When programs are type-safe, there is no need for builtin primitives to check the type of arguments before using them. This will reduce run-time overhead.
- *Produce more efficient code.* When a receiver is known to be an instance of one class only, method lookup can be replaced with a function call. This in turn enables other optimizations, e.g., inlining. Efficiency can sometimes also be improved even if the receiver could be an instance of more than one class [10].
- *Characterize polymorphic recursion.* In object-oriented languages, recursion has a deeper meaning, compared to other languages, since a function (a method) is identified not only by the name of the function, but also by the receiving object. Suppose an instance I sends a message m. Then we can differentiate between three kinds of recursion, in ascending order of generality:
  —Instance recursion: when the execution of m invokes the method for m in I again (perhaps transitively).
  —Class recursion: when the execution of m invokes a method for m in an instance of the same class as I.
  —Polymorphic recursion: when the execution of m invokes a method for m in some other instance (perhaps I). Thus, polymorphic recursion may not be recursive at all!

With type information, polymorphic recursion can sometimes be classified as non-recursive and therefore be allowed in RTT.

5. PRINCIPLES OF THE RUN-TIME ENVIRONMENT

5.1. Introduction

The run-time environment provides a platform for running RTT applications and consists of a set of nodes that are interconnected via a communication network.

The run-time environment contains both strictly hardware contained functions as well as software functions. However, many functions is a result of cooperation between hardware and software. In addition, a lot of the functions could either be implemented directly in hardware or software. Therefore, we do not separate the run-time environment into a hardware and a software part.

Most hardware components used in run-time environments today are designed to be used in systems with the goal of maximizing the average performance. As observed by Stankovic [22] these components have shortcomings when used in hard real-time hardware architectures. Therefore when designing efficient hard real-time run-time environments a lot of effort has to be spent on evaluating the predictability and the dynamics of existing components, design of new components, and study how the components fit together.

5.2. Functionality of the Run-Time Environment

The platform has to support execution of hard real-time modes according to a pre-run-time generated schedule. This schedule could consist of a number of subschedules, e.g., one subschedule for each node and one for the communication network. These subschedules have to run synchronized to behave as one system schedule. This requires that the clocks on each node are synchronized with a known accuracy. To minimize the damage in case of a timing fault, the deadlines of the use-cases in the current mode must be supervised.

Since RTT applications not only consist of hard real-time functionality the run-time environment has to provide a base for running the soft part of an RTT application. The soft part of an RTT application is as mentioned before scheduled on-line and should fulfill the timing requirements on a best effort basis. This feature should by no means jeopardize the temporal behavior of the hard real-time part of an RTT application.

The RTT framework put no demands on the topology of the communication network. For the hard real-time part of an RTT application, the communication services must support bounded end to end message transfer times. In analogy with the execution of an application there must exist communication services for soft real-time messages which does not interfere with hard real-time messages.

In object-oriented systems, as mentioned earlier, there are a number of basic features that could give variance in program execution and thus would give a poor utilization of the hardware resource. The two features which contributes the most to poor utilization are garbage collection and method dispatching. To be able to run real-time object-oriented applications, the run-time environment must have implementations of these features that are both predictable and have low variance in execution time.

5.3. Structure of the Run-Time Environment

If the run-time environment is divided into parts and each part has a strict functionality, for example, application, communication, method dispatcher, garbage collection, I/O, operating system, debugging, and monitoring parts, many benefits are provided. The following benefits was identified by Stankovic [22]

- It will be simpler to map an application to the resource structure because the application part could be isolated from unpredictable interrupts generated by the nondeterministic environment.
- The system would be more manageable due to the separation. It should be noted that each part must have its own resources to make the separation strict.

To be able to utilize any part efficiently there must be a small variation in execution time for each basic operation. For example the data moved in a move instruction could be accessible in cache memory or in the ordinary RAM.

This will give a big variation and when calculating the execution time for a real-time program the MAXTC has to be considered which will lead to poor utilization. A basic operation could also for example be a context switch.

To simplify the communication system it is preferable to use a broadcast bus topology. In such a topology, no relaying nodes have to be used in communication between nodes in the same network. Using such an approach, only one bus slot is used when sending one frame. This will simplify the tools that allocates bus bandwidth to the application.

In Section 6, we will present a prototype of the run-time environment.

6. AN RTT PROTOTYPE SYSTEM

6.1. Introduction

In this section we will briefly present some of the tools and solutions to the proposed models discussed earlier.

First, we will present a prototype of the run-time environment, thereafter a few words about the RTT compiler followed by a short description of the MAXTC Tool. Thereafter, we will present the principles of the configuration compiler, i.e., the resource mapping tool.

6.2. The RTT Prototype Run-Time Environment

6.2.1. Introduction. The run-time environment consists of a number of nodes. Each node is structurally partitioned into three separate units, one *application unit*, one *communication unit*, and one *time handling unit*. The two first units have their own resources such as CPU and memory. These two units communicate through a dual port memory. The time handling unit is implemented directly in hardware and thus also has its own resources.

The memory in the communication and application units are protected by a memory protection unit. This unit is developed especially for this architecture and will ensure that the tasks can only access memory to which they have been reserved rights. The reason for developing this unit is that commercial available memory management units have unpredictable timing behavior [22].

The communication and application unit also includes a real-time garbage collector [13].

6.2.2. The Application Unit. The application unit executes the application tasks and is responsible for the application I/O. The task execution platform is provided by the Rubus real-time operating system [1]. It contains guaranteed services (hard real-time) and best effort services (soft real-time). Rubus is divided into two executives, the hard real-time and the soft real-time executive.

- The hard real-time executive is based on the time-triggered execution paradigm and is dispatching tasks according to a pre-run-time generated schedule. It is also handles deadline supervision and takes care of deadline violations. The deadline control is made in cooperation with the time handling unit.
- The soft real-time executive is based on the event-triggered execution paradigm where a priority based scheduling policy is used. The hard real-time executive will always preempt the soft real-time executive when it is time to dispatch a hard real-time task.

6.2.3. The Communication Unit. The communication unit in each node is connected to a broadcast bus, in this case a CAN bus (ISO/DIS 11519, unit 1). This unit is responsible for the network communication and handles all messages sent to and from the application unit. It also handles group membership protocols [5].

The RTT communication protocol is implemented as an application layer on top of the data link layer of the CAN protocol. This protocol makes a distinction between hard and soft real-time messages. To fulfill timing requirements, hard real-time messages are scheduled pre run-time.

The protocol is based on the TDMA paradigm; i.e., the network bandwidth is divided into time slots. There is a maximum number of frames that can be sent in one bus slot. A frame corresponds to a CAN message. The nodes have a consistent view of the bus slots since there exists an approximate global time base.

TDMA based protocols usually allocate one bus slot per node. In its bus slot, the node can either send a frame or leave the bus slot unused. This leads to an inefficient use of network bandwidth. In the RTT communication protocol, several nodes can transmit frames in a bus slot according to the pre-run-time generated schedule. This is possible due to the collision avoidance arbitration mechanism provided by the CAN protocol.

Each node knows when all hard real-time frames have been sent every node listen to the traffic on the bus and has knowledge about the schedule. When all the hard real-time frames in a bus slot has been sent, soft real-time frames will be transmitted until the end of the bus slot. This makes it possible to get a high utilization of the network bandwidth.

6.2.4. The Time Handling Unit. The time handling unit includes separate timers for the communication and application units. The timers for each unit handles dispatching and deadline supervision of hard tasks and dispatching of soft tasks. Furthermore, this unit provides an approximate global time that could be read by the other two units. To be able to provide such a global time, a clock synchronization algorithm is implemented in the RTT communication protocol.

6.3. Compiler

The present version of the RTT compiler generates C-code which has to be compiled and linked with C-tools to generate a run-time system, as described in [9]. In this

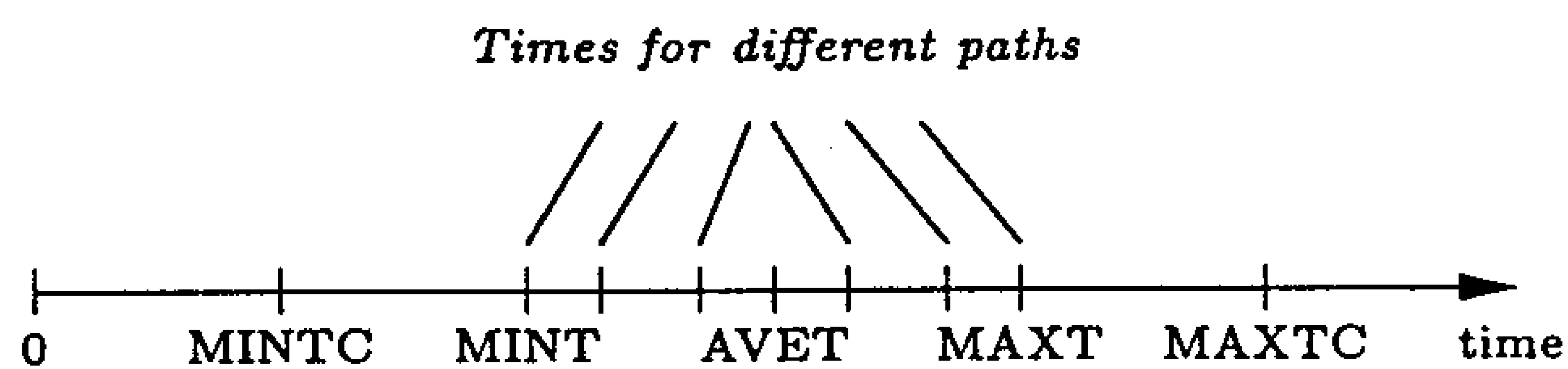


FIG. 9. Execution times for a program.

version, dynamic types are used, which may cause some problems, as mentioned in Section 4.3. The next version of the RTT compiler will attempt to solve these problems by using type inference. This compiler will also generate optimized assembly code, yielding a better utilization of resources.

6.4. *The MAXTC Tool*

The execution times for a program may vary depending on the path taken in the control graph. In Fig. 9, the following time measures are shown:

- MINT = real minimum execution time
- MAXT = real maximum execution time
- MINTC = calculated minimum execution time
- MAXTC = calculated maximum execution time
- AVET = $\Sigma_{i=1}^{n} T_i/n$ = pathwise average execution time of a program.
- OF = MAXTC/MAXT = overreservation factor.
- D = MAXT/MINT = dynamic factor.

The MAXTC calculation tool [9] calculates MAXTC. Naturally, the overreservation factor (OF) has to be as close to 1 as possible to avoid reservation of too large CPU resources in the run-time schedule. The tool also calculates MINTC. This number can for instance be used to assess the dynamic factor of a program. It also gives an indication of the spare capacity left for soft functionality. (Remember that hard real-time tasks are preallocated, while soft real-time tasks uses unallocated CPU-time and spare time from hard real-time tasks.)

Currently, there is a prototype of the tool which cooperates with the current dynamic type version of the RTT compiler. To be able to solve some problems with this implementation, the main one being high overreservation, the next version of the MAXTC tool will exploit the type information inferred by the compiler.

The method dispatch (or method invocation scheme) used in Smalltalk makes it difficult to calculate the execution time for a system before run-time. The reason for this is the linear search for methods which starts at the class of the receiver and goes up through the inheritance tree. When execution time is calculated, the time required for method dispatching must be predictable for every message. For this reason, a method dispatch table is used at run-time. The method dispatch table is built at compile time using an algorithm called modified two-way coloring (MTWC) by Huang and Chen [14]. With type information at hand most of the dynamically bound message sends could be statically bound. The reason for this is that the use of polymorphism is not used in the extent that one could believe [4].

6.5. *The RTT Configuration Compiler*

6.5.1. Introduction. In this section, we will present how the hard real-time part of an application is translated into a runnable application. As mentioned before, the hard real-time part is based on the time-triggered approach and scheduled prior to run-time. The translation process includes allocation of both CPU capacity for each node and bus bandwidth. The output from the translation process is a schedule for each mode and node, plus message space for the messages sent over the network. This translation is done by a tool called the RTT configuration compiler.

The configuration compiler requires an *architecture specification* and a *configuration specification.* The architecture specification lists the number of nodes and gives the characteristics of the nodes and the bus. The configuration specification describes the application as given by the designer, i.e., modes, use-cases, tasks.

The configuration compiler consists of two cooperating tools (Fig. 10): the *RTT communication handling tool* and the *RTT pre-run-time scheduler.*

The communication handling tool automatically inserts necessary system tasks, communication buffers, and mutual exclusion relationships to make it possible for the pre-run-time schedulers to find a schedule. Many pre-run-time schedulers only supports communication between tasks

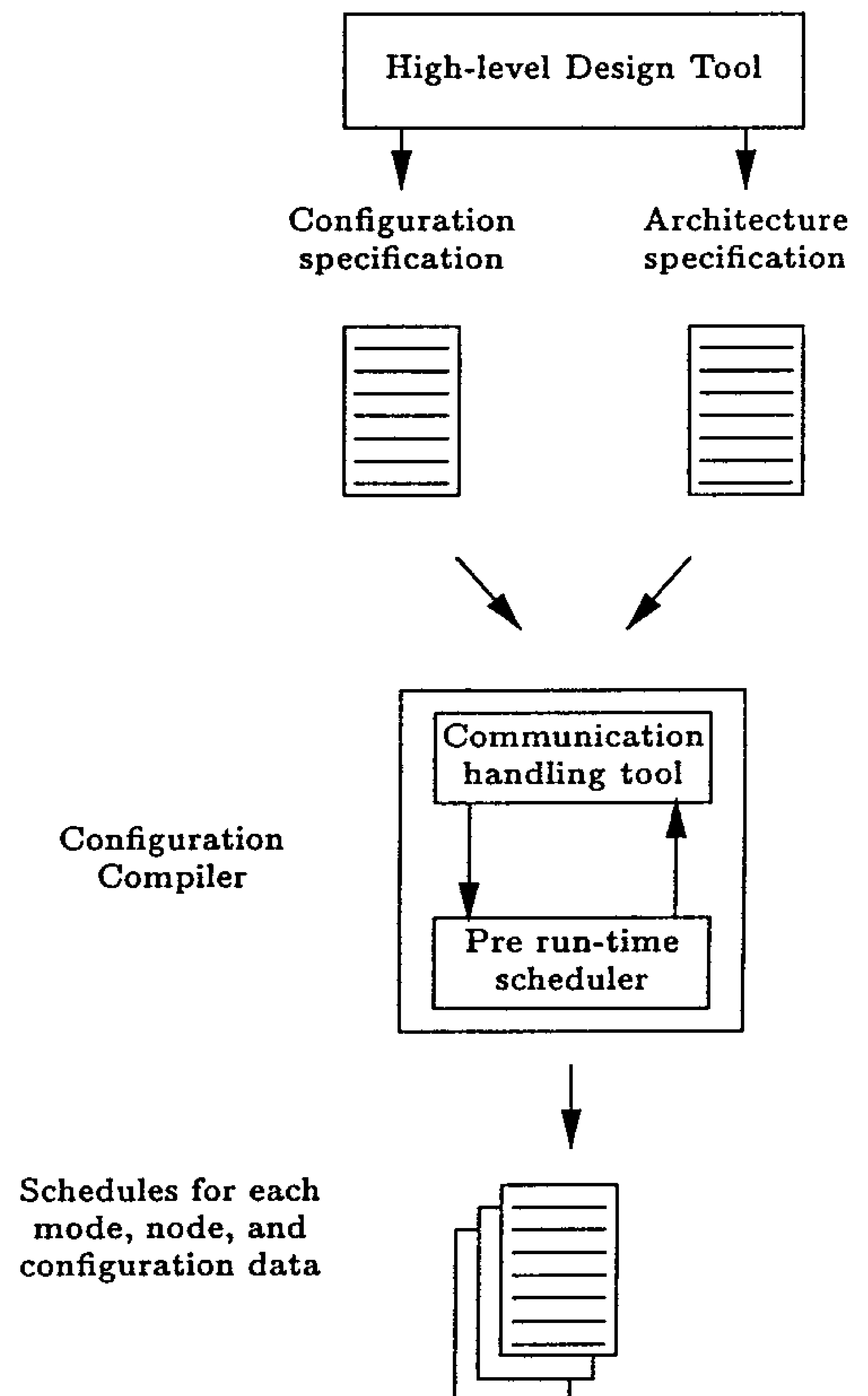


FIG. 10. The configuration compiler. A high-level graphical tool is used to produce the specifications for the compiler.

that have a precedence relationship [20, 16]. These schedulers cannot handle communication between task running with different period times, a desired property mentioned earlier.

In this section, we show how communication between tasks residing in different precedence graphs can be supported by using mutual exclusion relationships.

6.5.2. Architecture and Configuration Specification. To be able to transform an application to a resource structure, an architecture specification must be provided:

- The number of nodes in the system.
- The time between two consecutive points where the kernel may
  - —explicitly start the execution of a task;
  - —preempt a task;
  - —check if a task meets its deadline.
- The length in time of a bus slot. A message that is sent in one bus slot is available for the receiver in the next bus slot.
- The number of hard real-time frames that can be allocated to a bus slot by the scheduler.
- The accuracy of the clock synchronization.

The configuration specification includes synchronization and communication requirements for each mode of the application. This specification is generated from the programming tool which the user uses when designing an application. The structure of the configuration specification is very similar to the structure of an application; i.e., it includes modes, use-cases and tasks.

6.5.3. Communication Handling Tool. To support the communication requirements, system tasks are automatically installed by the communication handling tool. For example, assume two tasks communicating with each other such that the producer is a predecessor to the consumer in the precedence graph. Assume furthermore that the tasks are allocated on the same node. The communication tools will then install a system task as a successor to the producer and as a predecessor to the consumer. The role of the system task is to copy the contents from the producer's out-port to the consumer's in-port.

The reason for installing system tasks is that the communication will be transparent from the sender's point of view. So, for example, if we change the architecture to support redundant buses, we only have to change the system tasks.

Communication between two tasks can be performed between any two tasks that are defined in

- The same precedence chain and where the consumer is a successor to the producer. Multicast and broadcast are defined as a finite number of producer and consumer relations.
- The same precedence graph which have no precedence relationship.
- Different precedence graphs.
- The same precedence chain and where communication from a successor to a predecessor is required. This feature is useful to have when, for example, a control algorithm is implemented.

The communication can, of course, take part between tasks that are running on either the same node or on different nodes. It can either be buffered or unbuffered. When tasks share resources which do not allow concurrent access, the tasks must be defined to be mutually exclusive.

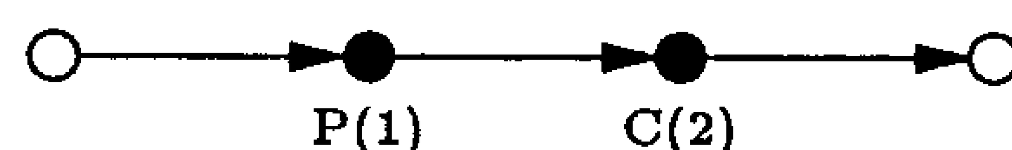
When a transformation is made, some system tasks are installed. A system task is from the run-time system's perspective a task which has special privileges. For example, a send task has the right to read from a task's out-ports. To be able to handle the communication requirements we have three types of system tasks:

- sendTask; this task reads a message from a predefined out-port of a task. Thereafter it passes the information to the communication subsystem. The information on where to read the information is passed to the system task as an argument when it is activated. The feature of passing information to the task is supported by the real-time kernel in use, Rubus [1].
- receiveTask; this task reads a message from the communication subsystem and stores the information at a predefined memory location. Normally, the memory location is a task's in-port.
- localSendTask; this task reads a message from a predefined out-port of a task and thereafter writes the message to a pre-defined memory location.

The translations for all the different cases can not be described here due to the limited space. Rather, we will describe the translation algorithm for two different cases. First, we will describe how the communication is handled by two tasks that are defined in the same precedence graph. Second, we will describe the communication between tasks that runs with different period times on different nodes. In the second example, we will also see how the mutual exclusion relationship can be used. The complete description for the different cases can be found in [7].

EXAMPLE 6.1. The producer and consumer is defined in the same precedence chain and the consumer is a successor to the producer in the precedence graph. The producer and consumer are allocated on different nodes (Fig. 11).

a) Specified precedence graph:



b) Transformed precedence graph:

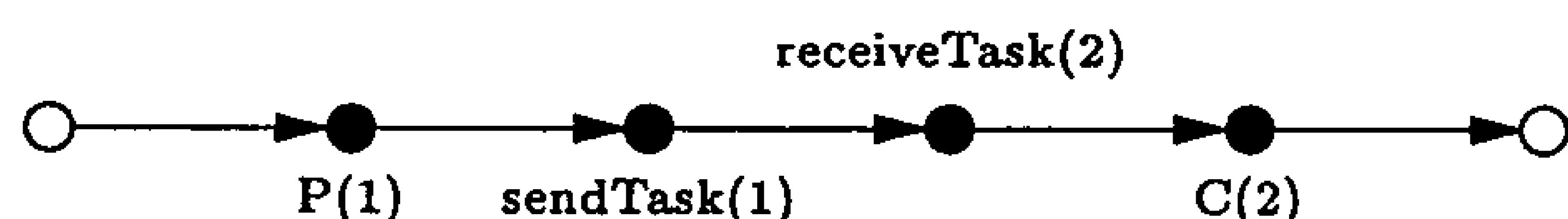


FIG. 11. The transformation of a precedence graph when the producer (P) and the consumer (C) are allocated on different nodes. The notation $C(n)$ means that the task C is allocated on node n.

Transformation:

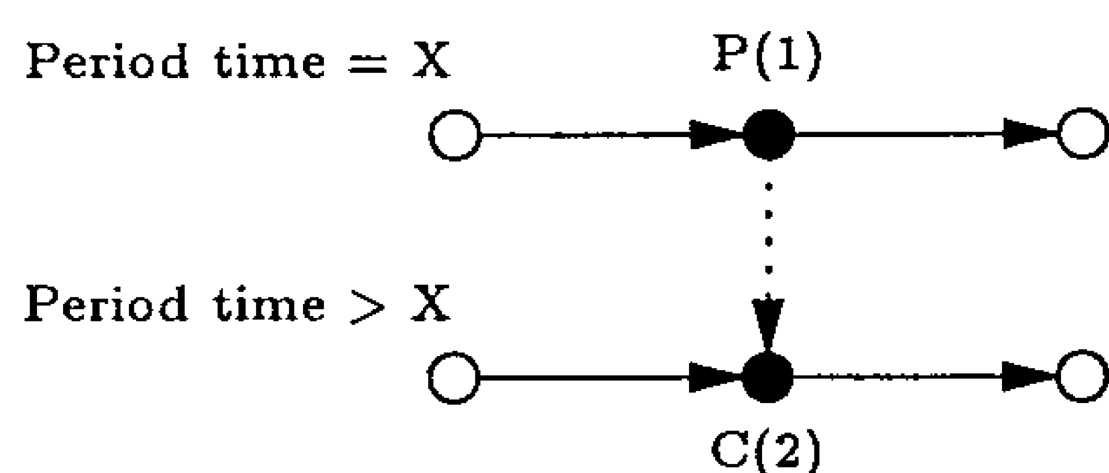
- Install a communication buffer to hold the temporary message on the producer's node.
- Install a send task as a predecessor to the consumer. The send task copies the contents from the producers outport to the installed communication buffer.
- Install a communication buffer on the receiver's node. This buffer holds the contents of the received message.
- Install a receive task on the consumer's node as a successor to the abovementioned send task, and as a predecessor to the consumer. The role of the receive task is to copy the received message from the communication buffer to the consumer's in-port.

EXAMPLE 6.2. The producer and consumer are defined in different precedence graphs and allocated on different nodes. The period time of the producer is less than the period time of the consumer (Fig. 12).

Transformation:

- Install a variable to hold a temporary message on the producer's node.
- Install a local send task as a successor to the producer. The send task copies the contents from the producer's outport to the installed variable.
- Install a communication buffer to hold the temporary message on the producer's node.
- Install a communication buffer to hold the message which was received by the consumer's node.
- Install a receive task on the consumer's node as a predecessor to the consumer. This task copies the message from the installed communication buffer to the consumer's in-port.
- Install a send task on the producer's node as a predecessor to the installed receive task. This task copies the message from the temporary variable to the installed communication buffer.

a) Specified precedence graphs:



b) Transformed precedence graphs:

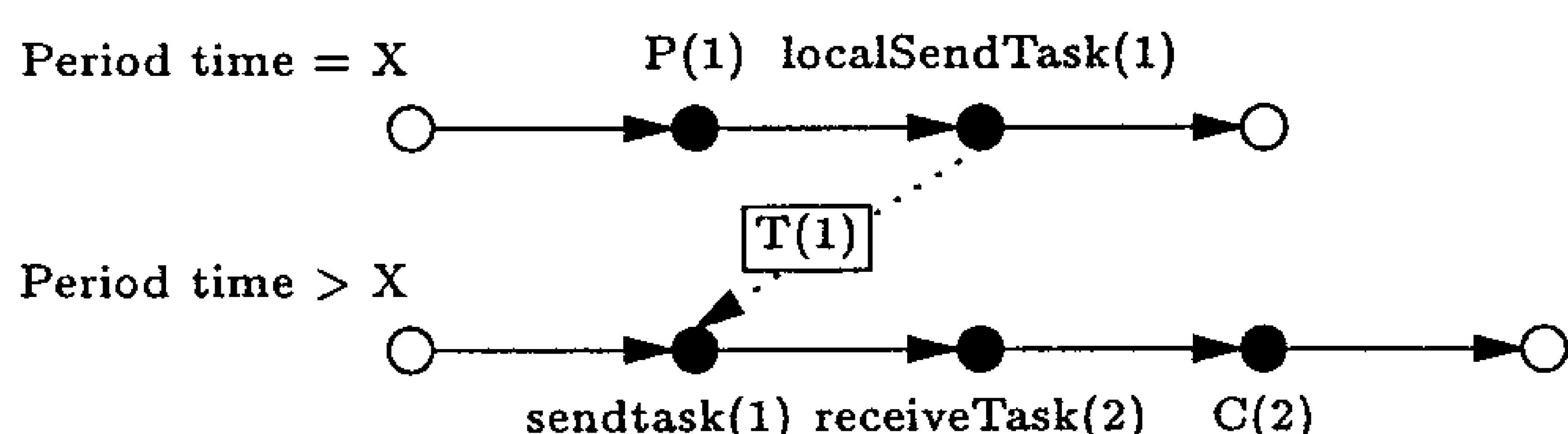


FIG. 12. The transformation when the producer's period time is less than the consumer's. Dotted lines indicate dataflow.

- Define a mutual exclusion relationship between the local send task and the send task.

The motivation to install a send task as a predecessor to the consumer at the producer's node is to minimize the traffic on the network. In Fig. 12, the box $T(1)$ depicts the temporary variable T allocated on node 1. The reason for defining a mutual exclusion relation between the local send tasks and the send task is to get a consistent message.

6.5.4. The RTT Scheduler. The scheduler tries to find a feasible schedule for the specification, using a heuristic search. The scheduler supports precedence and mutual exclusion relationships, but requires that tasks are preallocated to nodes. The mutual exclusion relationship is a binary relationship between two tasks (or two groups of tasks) which means that if one task is executing, the other task is blocked until the executing task terminates. This relationship is used when, for example, two tasks defined in different precedence graphs wants to access the same resource. A resource could for example be a shared object that is used to transfer information between two tasks. The scheduler can be configured to support either preemption or nonpreemption. The scheduler also supports bus allocation. The bus is scheduled by treating the bus as a processor and each frame is mapped to a specific type of task, named sendTask.

7. RELATED WORK

7.1. Introduction

This section compares RealTimeTalk with the following system architectures: DROL [23] (an object-oriented programming language for distributed real-time systems); MARS [16] (maintainable real-time systems); CHAOS [3] (concurrent hierarchical adaptable object system); ARTS [24] (a distributed real-time operating system) and DEDOS [12, 11] (dependable distributed operating system).

MARS and CHAOS describe a complete system architecture covering all steps from analysis to a running system. DROL includes an object model and programming model for distributed real-time systems. ARTS and DEDOS are focused on distributed operating system architectures. It should also be mentioned that no hardware details are covered and that all these systems, with the exception of MARS, are based on object-orientation; i.e., their language is based on C++ or an extension thereof. The MARS language, Modula/R, is a real-time extension of Modula/2.

7.2. DROL

In DROL, the synchronization and program logic is separated by meta-level objects. In RTT, this separation is done with precedence graphs. The advantages of our separation policy has been discussed before.

DROL can be used for applications with soft real-time

requirements while RTT is geared toward applications with both hard and soft real-time requirements.

The assumption that the system has no global time seems a bit strange and it is not clear how a server at another node can detect if a deadline will be missed or not. Additionally, it seems difficult to get an upper bound on delays throughout the system.

The time polymorphic invocation is a novel approach and it will be interesting to see if it is usable in real applications.

7.3. MARS

The major differences between MARS and RTT is the model of execution; in MARS, it is based on the process model, while RTT uses an object-oriented one. Furthermore, the MARS system supports only hard real-time, while RTT supports both soft and hard real-time.

There are, however, many similarities between the MARS system and RTT. For example, both systems separates the communication and synchronization from the code. In MARS, the precedence relationship between tasks are stronger because the precedence relationship includes both synchronization and communication. In RTT, these concepts have been separated to support communication between tasks residing in different precedence graphs and to allow implementation of feedback controllers in a straightforward way.

It should also be noted, that within the MARS system, redundant hardware aspects, dependability analysis, and testing is covered. These areas have not yet been covered in RTT.

7.4. CHAOS

A fundamental difference between CHAOS and RTT is CHAOS' support of adaptability. Adaptability means that the system can be changed during run-time to adapt to changes in the environment not known in advance. In comparison, an RTT system can only adapt to changes that are known in advance.

In the next generation of dynamic and adaptable systems, some parts of the system must be designed by preallocating all needed resources as in RTT. Furthermore, high-level features, as for instance planning, must be supported by an adaptable system. It seems that the dynamic behavior introduced in CHAOS will make it very hard to guarantee hard real-time requirements.

7.5. ARTS

The ARTS system is an event driven system, while RTT is a time driven system. The ARTS system can be seen as an architecture for testing different scheduling theories. The real-time object model is an interesting concept because it supports temporal encapsulation. Another difference, compared to RTT, is that the synchronization in ARTS is embedded in the code and thus inheritance anomalies could occur [2].

7.6. DEDOS

The DEDOS system is very interesting due to the support of both hard and soft real-time requirements. One of the big differences between DEDOS and RTT is the supported language; DEDOS is based on an extension of C++ called DEAL, while RTT is based on Smalltalk.

The communication style between the soft and hard real-time parts in DEDOS have also been used in RTT. However, in RTT, there is also support for sending events from the hard real-time part to soft real-time part. RTT also seems to be more focused on the analysis and design phase of a system. The explicit use of precedence graphs will hopefully make the system more easy to understand and maintain. In DEDOS, the precedence graphs are constructed from the program by a tool. When integrating the functions and synchronization within the code, inheritance anomalies could occur [2].

It should also be mentioned that DEDOS also includes support for fault-tolerance in both software and hardware.

8. CONCLUSION

We have presented the RealTimeTalk (RTT) system and how it supports important real-time areas such as synchronization, communication, distribution, timing requirements, modelling and programming. The RTT system basically consists of the following parts:

- Programming in the large.
  - —A model for designing both hard and soft real-time functionality.
  - —Interaction support between hard and soft real-time parts.
  - —Support for object-orientation as a mean for designing applications.
- A real-time language.
  - —A language based on Smalltalk with support for prototyping of applications.
  - —Real-time constructs to guarantee a predictable temporal behavior of an application.
- A run-time environment.
  - —A distributed hardware architecture.
  - —An operating system with support for both hard and soft real-time tasks.
  - —A communication system with support for both hard and soft real-time messages.
- Tools for mapping a specification to a resource structure.
  - —A compiler, with future support for type inference.
  - —A MAXTC tool, for calculating the maximum execution time for tasks.
  - —A configuration compiler, which synthesize the communication and synchronization requirements of an application.

The RTT approach has also been compared to other important real-time systems in the research community.

ACKNOWLEDGMENTS

This work was supported by the National Board for Industrial and Technical Development (Project 93-3180), the Västmanlands County Administrative Board, and fundings from Mälardalen University and the Royal Institute of Technology. We gratefully acknowledge help from Arcticus Systems AB on the run-time system.

REFERENCES

1. Arcticus Systems AB, Rubus OS Real-Time Operating System Tutorial. Technical report, Arcticus Systems AB Datavägen 9A, 175 62 Järfälla, Sweden, 1996.
2. L. Bergmans and M. Aksit, Composing synchronisation and real-time constraints. *The Object-Oriented Real-Time Workshop*, in conjunction with the 7th IEEE Symposium on Parallel and Distributed Processing, San Antonio, Texas, Oct. 1995, pp. 108–115.
3. T. E. Bihari and P. Gopinath, Object-oriented real-time systems: Concepts and examples. *IEEE Computer* (Dec. 1992).
4. E. Brorsson, C. Eriksson, and J. Gustafsson, RealTimeTalk: An object-oriented language for hard real-time systems. *Intl Workshop on Real-Time Programming WRTP'92, IFAC/IFIP*, Bruges, Belgium, June 1992.
5. C. Eriksson, M. Gustafsson, and H. Thane, A communication protocol for soft and hard real-time systems. *Euromicro Workshop on Real-Time 96*, 1996.
6. C. Eriksson, R. Hassel, K. Sandström, and L. Myrehed, A graphical design environment for the development of object-oriented hard real-time systems. *TOOLS Europe 95*, Paris, Mar. 1995.
7. C. Eriksson and K. Sandström, The translation of an application configuration to a runnable application by utilising a pre run-time scheduler. Technical Report CUS95RR02, Dept. of Computer Engineering, University of Mälardalen, Sweden, 1995.
8. A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Addison–Wesley, Reading, MA, 1989.
9. J. Gustafsson, Calculation of execution times in object-oriented real-time software—A study focused on RealTimeTalk. Licentiate thesis, 1994.
10. J. Gustafsson, K. Post, J. Mäki-Turja, and E. Brorsson, Benefits of type inference for an object-oriented real-time language. *Object-Oriented Real-Time Workshop*, San Antonio, Texas, Oct. 1995.
11. D. K. Hammer, P. Lemmens, E. Luit, O. S. van Roosmalen, P. van der Stok, and J. Verhoosel, DEDOS: A distributed environment for object-oriented real-time systems. *IEEE J. Parallel Distrib. Technol.* **2,** 4 (1994).
12. D. K. Hammer and O. S. van Roosmalen, An object-oriented model for the construction of dependable distributed systems. *2nd International Workshop on Object Orientation in Operating Systems*, 1992.
13. R. Hassel and K. Sandström, Garbage collection i realtid. Master's thesis, Univ. of Mälardalen, Sweden, 1993. [In Swedish]
14. S. Huang and D. Chen, Efficient algorithms for method dispatch in object-oriented programming systems. *J. Object-Oriented Comput.* (Sept. 1992).
15. H. Kopetz, Event-triggered versus time-triggered real-time systems. Lecture Notes in Computer Science, Vol. 563, pp. 87–101. Springer-Verlag, Berlin/New York, 1991.
16. H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft, and R. Zainlinger, Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro* **9,** 1 (Feb. 1989), 25–40.
17. N. Leveson, *Safeware, System Safety and Computers*. A Wesley, Reading, MA, 1995.
18. A. K. Mok, Fundamental design problems of distributed syst the hard-real-time environment. Ph.D. thesis, MIT, 1983.
19. P. Thijssen, P.D.V. van der Stok, and L. Somers, Formal veri and simulation of a real-time concurrency protocol. *Inter Workshop on Responsive Computer Systems*, 1992.
20. K. Ramamritham, Allocation and scheduling of complex tasks. *10th Int. Conf. on Distributed Computing Systems*, 1 108–115.
21. J. A. Stancovic, Misconceptions about real-time computing *Computer* **21** (Oct. 1988), 10–19.
22. J. A. Stankovic, The Spring architecture. *Euromicro Works Real-Time*, June 1990, pp. 104–113.
23. K. Takashio and M. Tokoro, An object-oriented language fo uted real-time systems. *OOPSLA '92*, Vancouver, 1992, pp. 1
24. H. Tokuda and C. W. Mercer, ARTS: A distributed real-time *Oper. Systems Rev.* **23,** 3 (1989).
25. M. Törngren, Modelling and design of distributed real-time applications. Ph.D. thesis, Dept. of Machine Design, Royal of Technology, Stockholm, Sweden, 1995.

CHRISTER ERIKSSON received a B.Sc. in mathematics fro sala University in 1988 and a licentiate degree from the Royal of Technology, Stockholm, Sweden in 1994. He worked for ABB tion AB from 1984 until 1988, primarily on the run-time system Master. He is currently a lecturer at Mälardalen University. His interests are design of real-time systems, object-oriented progr distributed architectures for real-time systems, and real-time o systems.

JUKKA MÄKI-TURJA received a B.Sc. in computer scien Mälardalen University in 1993. Since 1994, he has been a Ph.D. at the University. His primary research interests are in langua compilers, especially program analysis. He is currently involve RealTimeTalk project, working with the RTT language and its ty ence system.

KJELL POST received an M.Sc. in computer engineeri Linköpings Universitet in 1987 and a Ph.D. in computer science University of California, Santa Cruz in 1994. He is currently a pro Mälardalen University. His research interests are implementat analysis issues for programming languages.

MIKAEL GUSTAFSSON received an M.Sc. in electrical eng from the Royal Institute of Technology, Stockholm, Sweden Between 1984 and 1991, he was employed by ABB Automatio a designer of man-machine communication software used in t Master process control product family. Since 1991, he has be Mälardalen University, where he is currently a lecturer. His research interests covers computer architectures, operating syste data communication protocols for hard real-time systems.

JAN GUSTAFSSON is a senior lecturer at Mälardalen Un where he currently is the Head of the Department of Comput neering. He received a B.Sc. in mathematics, astronomy, and c science in 1974 from Uppsala University and a licentiate degre chine elements, computer controlled mechanics in 1994 from th Institute of Technology in Stockholm. From 1975 to 1985, he ABB (then ASEA), working with development of real-time i control systems. Since 1985, he has been with Mälardalen Un where his main topics are programming methodology, object- programming, and real-time systems.

KRISTIAN SANDSTRÖM received a B.Sc. in engineering f lardalen University, Västerås, in 1995. Since then, he has been in the Department of Computer Engineering at Mälardalen U

as a research engineer. His primary research interests are software models for real-time systems, design of real-time systems, object-oriented programming, and scheduling in real-time systems.

ELLUS BRORSSON receive a M.Sc. in mechanical engineering from the Royal Institute of Technology, Stockholm, Sweden in 1989. Since then, he has been with the Mechatronics group in the Department Machine Design at the Royal Institute of Technology, where he has be working as a teaching fellow and attending the Ph.D program. Recent he became a lecturer in the Computer Engineering Department at D larna University. His primary research interest is real-time systems a languages for real-time.

Received November 1, 1995; revised February 1, 1996; accepted April 25, 1996