University of Groningen **Faculty of Mathematics and Physical Sciences**

Department of Computing Science

Simulation of a Railroad Crossing: a Test Case for Artie

G.J. te Winkel

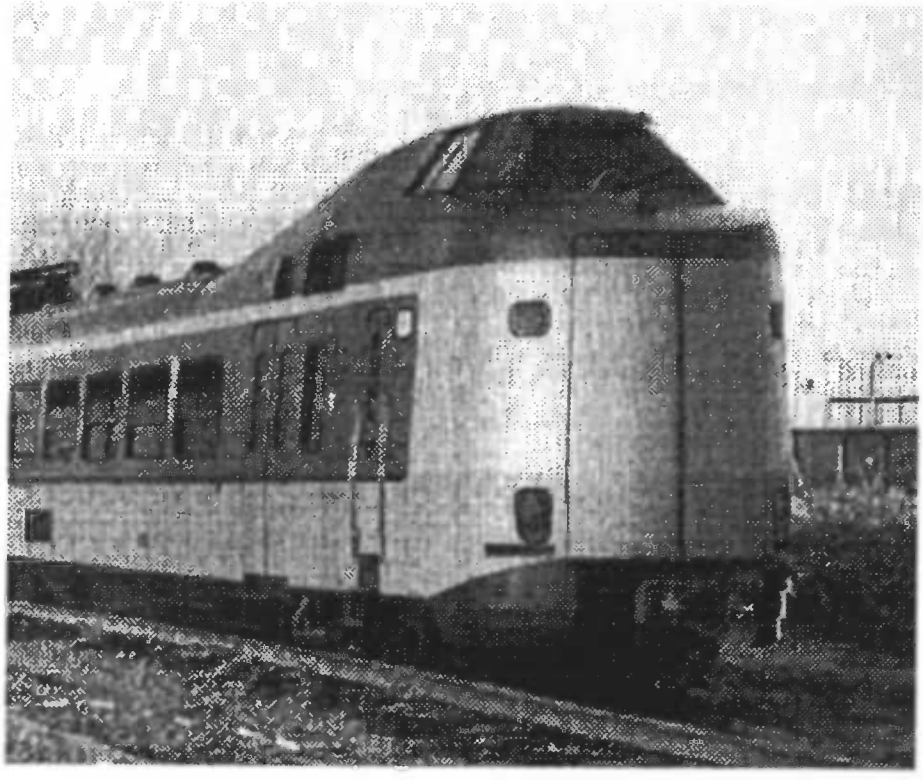
begeleider: Dr. R.M. Tol

September 1996

RuG

Simulation of a Railroad Crossing:

a Test Case for ARTIE



G.J. te Winkel

September 1996

Acknowledgements

This report is written within the framework of my Master of Science project from the Rijks*universiteit* Groningen, Department of Computing Science. Ronald Tol, my supervisor, introduced me in the world of ARTIE. Throughout the research project he helped me going the right direction. He also corrected this report and gave me valuable tips for improving it. I am very grateful for this and I would like to thank him for all this work.

I thank Hindrik Hettema for helping me with the implementation of ARTIE and designing the communication protocol for the simulator.

Special thanks go to Margreet who helped me during my entire study. She supported me in these years. Especially the last two years were not very easy. She also gave me hints to improve this report, for which I am very grateful.

Groningen, August 31, 1996
Gert te Winkel

Summary

ARTIE (**A**rchitectecture for hard **R**eal-**TI**me **E**nvironments) is a new architecture for real-time systems. ARTIE consists, among other things, of two processors. The kernel processor runs the operating system and on the task processor the application is executed. In this report ARTIE is evaluated by simulating a typical real-time application, a controller for a railroad crossing.
The application is developed with the use of the CASE-tool EPOS. In EPOS the application is designed stepwise. At the start the problem statement is given. The development process ends with a complete design, an application consisting of tasks that can be executed on the task processor of ARTIE.
With a formal method, wp-calculus, the functionality of the tasks are verified. Because wp-calculus is not specially suitable for the specification of real-time systems, Statecharts are also used for the verification. It is also proven that the set of critical tasks, tasks which should never exceed their deadline, is feasible.
The simulator consists of the implementation of ARTIE and a simulator of the environment of the railroad crossing. During the simulation sessions information about ARTIE's behaviour is collected. with this information ARTIE is evaluated.

Contents

Chapter 1

Introduction

Nowadays, people more and more depend on computers. Moreover, even in situations where lives of human beings can be endangered, computer systems are used. Such systems are called safety-critical computer systems. It is undesirable when normal computer systems fail, but it is really dangerous when a safety-critical computer system turns out to be unreliable. To guarantee the trustworthiness of such a computer system, a new Architecture for hard Real-TIme Environments (ARTIE) was developed [HS91].

In this report the behaviour of ARTIE is tested and evaluated. A typical real-time application is used as test case, the controller for a railroad crossing. This is a safety-critical application, and therefore it is specially important that design errors are avoided. EPOS, a CASE-tool particularly suited for real-time applications, can help the system designer to develop the application in a structured way, and so to avoid design errors. Therefore, EPOS is used to design the application. In Chapter 2 the description is given of the steps that were taken during the development.

After the development of the controller with EPOS, the tasks are formally verified in Chapter 3. It is proven that the application developed satisfies with the specification of the controller. In this chapter also the attributes for the tasks are determined, for example which task is critical and which not. Then it is proven that the set of critical tasks is feasible.

In Chapter 4 we will look at the simulator that will be used to test ARTIE. The different parts of the simulator are discussed as well as the communication between those parts.

In Chapter 5 the data that should be collected to examine ARTIE's behaviour are discussed. After that, several simulation sessions are done. Normal and exceptional situations are simulated. Finally, the results of the simulation sessions are presented and discussed.

The conclusions are presented in Chapter 6. In the appendices the complete EPOS-S design, the code of the tasks and a handout for the simulator can be found.

In this first chapter a brief review of the most important parts and features of ARTIE is given. The application is also introduced.

1.1 A brief description of ARTIE

ARTIE is a two-processor system. It consists of a task processor for the application program and of a co-processor for the real-time operating system kernel. The kernel provides, among other things, event handling and scheduling of tasks.

1.1.1 Event handling

The environment of the application can be considered as a set of (external) events. When changes take place in the environment of the application, an event is generated. The real-time kernel of ARTIE handles these events by giving an appropriate reaction on it. The reaction consists of executing one or more programs, so-called tasks. The set of associated tasks for every event is provided in a table by the application programmer. We call this table the Event Table.
Another type of events are internal events, or supervisor calls (SVC's). These events are used for communication between the task processor and the ARTIE kernel.

1.1.2 The task model

The application that runs on ARTIE is modelled as a set of tasks. A task is a program that can be executed on the task processor. Tasks can be triggered by events or by other tasks. For executing tasks, ARTIE needs some information about them. Therefore, every task b has the following task attributes that give these information.

- $b.r$, the release time of the task, i.e. the time when the task was made ready.
- $b.s$, the start time of the task, i.e. the time a task was made running.
- $b.e$, the worst-case execution time of the task, i.e. the maximal time needed to execute the program code of the task. This value depends on the value of the program code of the task.
- $b.d$, the deadline of the task, i.e. the time when a task has to be finished execution.

The application programmer stores the values of $b.r$ and $b.d$ in the Task Table. If the kernel needs the attributes, it can retrieve them from this table. The kernel determines the value of $b.s$ and $b.e$ is calculated during runtime and depends on the CPU of the task processor.

1.1.3 Scheduling

If more than one task is ready for execution at the same time, a choice has to be made which task to execute first. These decisions are made by following a certain scheduling policy.

In real-time computer systems the scheduling policy is of great importance. A wrong decision can result in a task missing its deadline. The consequence is that the task will not finish in time, which can lead to unforeseen or even dangerous situations. ARTIE uses a variant of the 'earliest-deadline-first' (EDF) strategy: earliest-critical-deadline-first (ECDF). Scheduling according to EDF assigns the task with the earliest deadline to the task processor. It has been shown that this strategy is optimal , i.e. the strategy will always produce the schedule where all tasks meet their deadline, whenever there exists such a schedule. With this scheduling policy, it is possible that an important task misses its deadline, because a less important task with a smaller deadline was put earlier in the schedule. Therefore, ARTIE distinguishes two levels of importance, critical and non-critical tasks. Critical tasks are always scheduled before non-critical task s. Within a level, the earliest-deadline-firs t policy is followed.
The developer of the application has to ensure that the set of critical tasks is feasible: every task in this set always has to meet its deadline. To implement these two levels of importance, tasks have a fifth attribute: *criticalness*. This attribute indicates whether a task is critical or not. When a non-critical task exceeds its deadline, it is simply discarded.

1.2 Application: The railroad crossing

A typical example of a real-time application is a controller for a railroad crossing. In [HJL93] such an application was proposed as a benchmark for the comparison of different formalisms to describe real-time systems. We use this application to test and evaluate ARTIE.

Consider a railroad with two tracks and a railroad crossing. Trains may travel in either direction, cars only on the right half of the road. If a train approaches the crossing, the gates that prevent cars to enter the crossing are lowered. The gates will raise as soon as the last train has left the crossing. This application provides an extra safety feature, namely when cars got stuck on the crossing while a train approaches, the signal light for trains is turned to red and the train will stop before it crashes on the car[1].

Several sensors are used to detect changes in the environment (see Figure 1.1). Four sensors detect a train approaching the crossing, four sensors detect leaving trains, two detect cars entering the crossing and two sensors detect cars leaving it. Furthermore,

[1]This last feature is not part of the problem as described in [HJL93]. We have added it to make the application slightly more interesting.

each gate has two sensors: one detects when the gate is closed and one detects when the gate is open.
When a sensor detects something, a signal is sent to the controller of the railroad crossing. These signals are treated as (external) events. Note that a signal is sent from the gates only if both gates are closed or open. If an event is generated by the

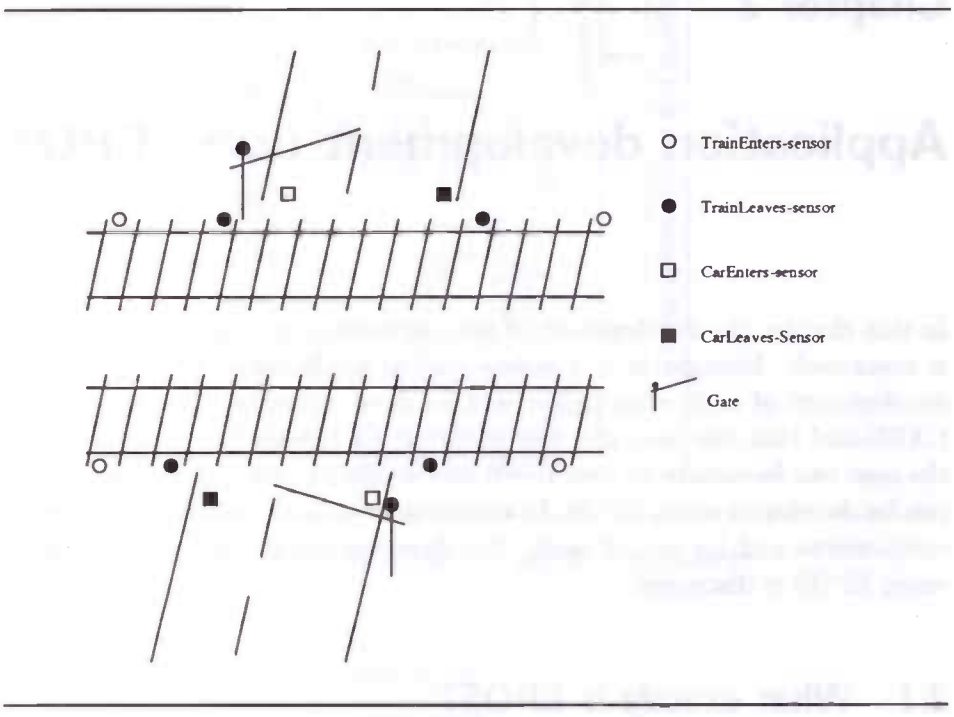


Figure 1.1: The Railroad Crossing

environment, the controller has to react upon each of these events properly. If a train is detected approaching the crossing, the controller reacts to this event by sending a signal to the gates. The gates close and cars can no longer enter the crossing. If the gate was already lowered, the controller does not have to take any action.
At the moment the gates are closed, a signal is sent to the controller. If a car is still on the crossing while the gates are closed, the controller turns the signal light for the train to red and the train is forced to stop. To be sure that the train has stopped before it has entered the railroad crossing, the sensors that detect approaching trains, must be far from the crossing, far enough to close the gates and to stop a train before the crossing. The train is allowed to enter the crossing if n o cars are on the crossing. The controller has to raise the gates when the last train has left the crossing.
To guarantee that the crossing is safe, the controller has to know the number of cars are on the crossing. Therefore, every time a car enters or leaves the crossing the controller is notified.

Chapter 2

Application development using EPOS

In this chapter the development of the application, a controller for a railroad crossing, is described. Because it is a safety-critical application, we have to ensure that the development of such a controller will be done correctly. Therefore, we use EPOS[1] a CASE-tool that can help the user to develop a system in a structured way. In EPOS, the user can formulate its own needs and define its own assumptions. Then the system can be developed using EPOS. In this chapter a brief description is given of EPOS, its components and its use. Finally, the development of the railroad crossing controller using EPOS is discussed.

2.1 What exactly is EPOS?

EPOS (Engineering and Project Management-Oriented Specification System) is a software tool designed to support system development and project management activities. Its aim is to release the system developers from routine work, enabling them to spend more time on development activities [Lau90].

EPOS uses an iterative version of the 'waterfall model', with a verification and validation phase after each block (see Figure 2.1). This model is widely used in practise. EPOS consists of:

- a user-friendly dialogue system, EPOS-C
- three specification languages, viz.:
  - ◇ EPOS-R: for requirements engineering

[1] EPOS is marketed by:

GPP, Gesellschaft für prozeßrechnerprogrammierung mbH
Kolpingring 18a
82041 Oberhaching bei München

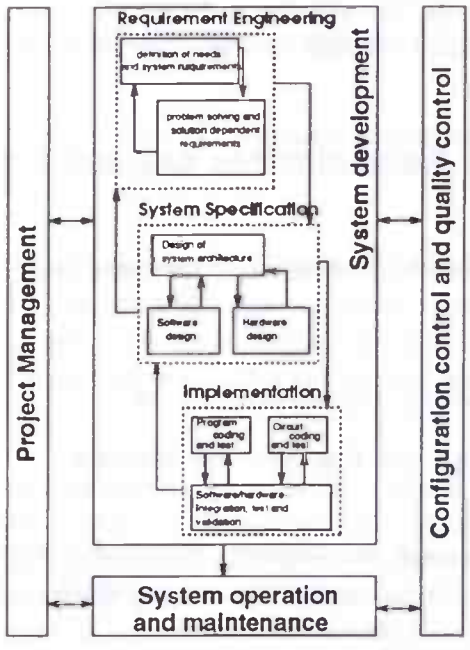


Figure 2.1: Overview of the EPOS system and its components

  - ◇ EPOS-S: for system specification
  - ◇ EPOS-P: for project management
- the project database
- several support tools, viz.:
  - ◇ EPOS-A: for analysis
  - ◇ EPOS-D: for documentation
  - ◇ EPOS-M: for management support
  - ◇ a Design Method support tool
  - ◇ a Code Generation tool

The controller for a railroad crossing is rather a small application. Therefore, we did not use all the components of EPOS, for example, we did not use EPOS-P. In the next sections we consider the development of the application using EPOS-R and EPOS-S.

In the next section of this chapter the design of a controller for a railroad crossing is started using the specification language EPOS-R. In Section 1.2 we have described the protocol of the railroad crossing controller. We have presented the events that can occur and the reaction of the controller on these events. We use this description of the

controller for the first step in the design. Therefore, we have not used the requirement engineering extensively. We will give some (small) examples of how it can be used. We also indicate some links between EPOS-R and EPOS-S.

2.2 EPOS-R: Requirements engineering

EPOS-R is a semi-formal language for expressing requirements. It is divided into two parts:

- problem statement: the needs document,
- solution concept: the functional specification.

In the problem statement, requirements and constraints are formulated which have no concern with the solution concept but are associated solely with the desired functionality of the system to be developed.
Within the solution concept, specifications are entered to describe the solution concept selected to cover the specified requirements and constraints and the reason for the choice. At the same time, a specification of requirements dependent on the solution concept is made.
The basic structure is a decadic outlining scheme, consisting of chapters and sections. A scheme can be defined by the user in each of the two parts. In the sections the following can be included: informal texts as part of the EPOS database and texts and graphics from files outside of the database, the definition and referencing of terms used, and the definition, substitution and fulfilment of 'identifiable requirement components' (`REQUIREMENT` and `CONSTRAINT`), using informal text, decision tables and state processes.
In order to avoid confusion between chapters and sections of this report and chapters and sections of the EPOS-R scheme's we use the symbol § for the latter.

2.2.1 Requirements and constraints of the railroad crossing

Before we start the requirement engineering, we first give some assumptions made about the railroad crossing.

- The railroad has two tracks, so not more than two trains can be present at the same time.
- The maximum number of cars that can be on the crossing at the same time is six.

- Trains and cars will always be detected (i.e. signals from sensors sent to the controller always arrive).
- Closing or opening the gates is always detected (i.e. signals from these sensors always arrive).
- The time passing between two trains going in the same direction is at least three minutes.

It is possible that situations occur which are different from the 'normal' situations as described in Section 1.2. The controller has to handle these situations in such a way that the crossing remains safe, i.e. that no cars are on the crossing when a train enters it. When an exceptional situation occurs the proper reaction of the controller is given in the next part.

If a failure in the power supply occurs, the controller can not control the gates anymore. Therefore, we assume that a gate is kept open with an electromagnet. When the power is lost, the gates close due to gravity, so the situation remains safe.
If the gates are closed and a car crashes through them, the situation becomes unsafe. The train has to stop to avoid a collision with the car. Because the gates are broken the train can not continue its way after the car has left the crossing. The train can only be restarted and pass the crossing safely with the help of the traffic controller of the railroad company. When the gates are repaired the automatic railroad crossing controller can take over the job again. Note that a collision can not always be avoided, the car can crash through the gates just in front of the train, so the train has no time to stop in time.

2.2.2 Problem statement

In the needs document we state the constraints and requirements for the railroad crossings controller. Figure 2.2 shows a possible outline for this document. In §1 of the needs document we can explain what a controller of a railroad crossing is and what requirements the controller has to fulfil. In §2 we can give the requirements of the controller in normal situations. §3 can contain the requirements of the system if the situation is not normal. All this information and requirements can be found in the Sections 1.2 and 2.2.1 of this report, just as the definitions of normal and not normal situations.
In §4 we can give the requirements that are asked when the controller is implemented on ARTIE. For example, ARTIE requires that the application is written as a set of tasks.

The requirements and constraints can be defined using the keywords `REQUIREMENT` and `CONSTRAINT`, respectively. For example, from the description of the controller in Section 1.2 we know that the railroad crossing always has to be safe. Figure 2.3 shows

```
1.   "Aim of the railroad crossings controller."
1.1. "Environment"
1.2. "Goal of the system"

2.   "System processes in normal situations"

3.   "System processes in exceptional situations"

4.   "Implementation for ARTIE"
```

Figure 2.2: Part of the scheme of the problem statement

how this requirement can be stored in the project database. Every requirement and constraint has an identification number and a revision number. So, this is the first version of requirement 101. The text between <> indicates a category. These are used to assign the component to a variety of categories. This allows the system designer to select and sort the identifiable requirement components into any desired group. It is

```
REQUIREMENT 101 (1)
 <Safety>
"The railroad crossing has to be safe."
```

Figure 2.3: Definition of a requirement

not clear what 'safe' exactly means, therefore we substitute the requirement by new ones. With safe we mean that collisions between a car and a train have to be avoided. So, if a car is on the railroad crossing or the gates are not down, the train is not allowed to enter the railroad crossing. The substitution and the new requirement are given in Figure 2.4. In large projects, where a system is developed by several people, it is necessary that definitions of terms used are unambiguous. EPOS offers the user the possibility to define terms and store them in the project database. When a term is put into the database, the user can search terms, mark them and set up a corresponding index. In EPOS-S the user can even create design objects with the same name and thereby establish a relation between EPOS-R and EPOS-S. In Figure 2.5 we give an example of the definition of the term railroad crossing.

```
SUBSTITUTE:
   REQUIREMENT 101 (1) BY REQUIREMENT 102 (1)
                          REQUIREMENT 103 (1)

REQUIREMENT 102 (1)
 <Safety>
"A train may only enter the crossing if the gates
 are closed."

REQUIREMENT 103 (1)
 <Safety>
"A train may never enter the crossing if a car
 is present."
```

Figure 2.4: Substitution of a requirement and the definition of the new requirements

```
TERM railroad crossing
 <General>
"A place on a railroad track where cars can cross.
 This place is protected by gates on the road to
 avoid accidents between cars and trains."
```

Figure 2.5: Definition of the lexicon term 'railroad crossing'

2.2.3 Solution concept

In the solution concept specifications are entered to describe the solution strategy selected to cover the specified requirements and constraints and the reason for the choice. At the same time, a specification of requirements dependent on the solution concept is made.

The designer can develop the solution concept with a top-down approach using stepwise refinement. §1 of the document contains a rough analysis of the system, a possible solution and the description of the various parts of the solution. The user can also give alternative solutions. The next § is a refinement, every part of the solution will be worked out in more detail. The designer can go on until he thinks the solution is sufficiently detailed. Figure 2.6 shows a possible outline of the solution concept for the railroad crossing controller. The controller for the railroad crossing can be considered

```
1.   "Solution concept for the controller for a
      railroad crossing and identification of the
      parts of the solution."
1.1. "Rough analysis of the controller's
      behaviour."
1.2. "Determination of the solution."
1.3. "Description of the parts of the solution. "

2.   "Definition of the parts of the solution."
2.1. "Detection of approaching trains."
2.2. "Detection of leaving trains."
2.3. "Detection of entering cars."
2.4. "Detection of leaving cars."
2.5. "Detection of closed gates."
2.6. "Detection of raised gates."
```

Figure 2.6: Part of the scheme of the solution concept

as a process that is waiting for an occurrence of one or more events. A possible solution for the controller is an infinite loop that waits for one or more events. In §1.3 we can describe which events the controller can expect. The sensors (see Section 1.2) can generate the following events: TrainEnters, TrainLeaves, CarEnters, CarLeaves, GateClosed and GateRaised. The names of these events explain when they occur.
In §2 of the solution concept we explain what should happen when these events occur. In §2.1 a description is given of a possible solution for handling approaching trains (or, in other words, the occurrence of TrainEnters). Figure 2.7 gives a possible solution.

```
2.1. "Detection of approaching trains."
     "If the gates are not already down or moving
      down the controller has to sent a signal to
      close the gates. If the gates are not closed
      within a certain time or if a car is still on
      the crossing while the gates are closed, the
      train has to stop."

     FULFILS: REQUIREMENT 102 (1)
     FULFILS: REQUIREMENT 103 (1)

     "If the railroad crossing is safe the train is
     allowed to enter it."
```

Figure 2.7: A possible solution for detection of approaching trains

We now can see that §2 fulfils the two requirement we have defined during the requirement engineering. This is indicated by the keyword FULFILS followed by the component that is fulfilled.

The parts §2.2 to §2.6 of the solution concept can be treated similarly.

2.3 EPOS-S: System design

During the system specification in EPOS-S, the problem-solving concept is transformed into a system. To achieve a reliable design a systematic approach needs to be used. The system is designed following a top-down approach of stepwise refinement. The total system is decomposed in a number of sub-systems. The sub-systems into sub-sub-systems. Until the complete system has been designed.
Each (sub-)system the components are described using the EPOS-S design objects. These are identified with keywords or (in a graphic representation) by symbols. In EPOS-S seven design objects exists:

- ACTION MODULE: describes subsystems,
- ACTION: describes activities in which data are operated on,
- DATA: description of information,
- INTERFACE: description of data exchange,

- EVENT: description of events which influence the order of actions,
- DEVICE: description of units and devices which perform actions or interface functions,
- CONDITION: data-dependent conditions which influence the control flow of the actions.

Every EPOS-S design object has the same basic structure. It begins with a keyword to indicate what type of object it is (e.g., ACTION, DATA, EVENT) and an arbitrary name (identifier) to identify the design object. There are three optional parts: the description part, the decomposition part and a part that defines relations to other design objects. The definition of the design object ends with the object type identifier plus a suffix to indicate termination (e.g., ACTIONEND, DATAEND, EVENTEND).

From §1 of the solution concept we know that the controller can expect the occurrence of six events. In EPOS-S an action can be triggered by an event with the following construction: WAIT FOR *event* THEN *action* WAITEND. The handling of the six events has to be done parallel, because events can occur at the same time. So, a possible action module is:

```
ACTION Controller.
DECOMPOSITION:
  WHILE TRUE
  DO PARALLEL
     ( WAIT FOR TrainEnters THEN TrainIn WAITEND,
       WAIT FOR TrainLeaves THEN TrainOut WAITEND,
       WAIT FOR GateClosed THEN GateIsDown WAITEND,
       WAIT FOR GateRaised THEN GateIsUp WAITEND,
       WAIT FOR CarEnters THEN CarIn WAITEND,
       WAIT FOR CarLeaves THEN CarOut WAITEND).
  OD
ACTIONEND.
```

In the design of the event objects, the designer has to indicate to which category the event belongs. These categories are: interrupts, cyclic events, events which occur only at certain moments and events which are used for synchronising a 'rendezvous'. In the case of the controller, all six events are interrupts.
If an event triggers an action this is indicated by the keyword TRIGGERS, followed by the action it triggers. Below we give an example of the event-object TrainEnters.

```
EVENT TrainEnters.
INTERRUPT.
TRIGGERS: TrainIn.
EVENTEND.
```

The next step is to design the six actions that are triggered by the six events. We show how the action TrainIn is created. From §2.1 of the solution concept we know that when an approaching train is detected, the gates have to be closed if the detected train

is the first train in the critical region. So, the controller has to know how many trains are present. In the variable NumTrains we store the number of trains. The variable NumTrains is an integer (in EPOS-S: FIXED) and because we have a two track railroad its range is between zero and two.

```
DATA NumTrains.
TYPE: FIXED.
RANGE: 0 -> 2.
DATAEND.
```

In TrainIn first the number of trains is increased by one. This is done in the action IncTrains. Now we have to ensure that the gates are closed and the crossing is empty when the train wants to enter it.
If another train was detected already (NOT NumTrainsIsOne), the train can continue its way. If not, the controller has to close the gates. This is done by the action Lower. If it takes too long before the gates are closed, longer than MaxLowerTime, the train has to stop, because otherwise there is a chance that the train enters the crossing while the gates are not closed. This action is undertaken by LowerGateTimeout. When the gates are closed, the action GateIsDown is executed. This action restarts trains that were stopped, when the crossing is empty.
If the crossing is not empty when the gates are closed, the action CarSafetyMeasure is executed. The action turns the semaphore for trains to red, so the trains will stop.

This results in the following action-objects TrainEnters, GateIsDown and CarSafetyMeasure:

```
ACTION TASK TrainIn.
DECOMPOSITION:
  IncTrains;
  IF NumTrainsIsOne THEN
    SET (GateClosed);
    Lower;
    WAIT FOR GateClosed
    WITHIN MaxLowerTime THEN
      GateIsDown;
      IF NOT NumCarsIsZero THEN
        CarSafetyMeasure;
      FI
    ELSE
      LowerGateTimeout;
    WAITEND
  FI.
TRIGGERED: TrainEnters
ACTIONEND.
```

```
ACTION TASK GateIsDown.
DECOMPOSITION:
  STOP (LowerGateTimeOut);
  RESET (GateClosed);
  IF NumCarsIsZero THEN
    STOP (CarSafetyMeasure);
  FI
  IF SemaStatIsRed
    THEN RestartTrain; FI.
CODE: "GateStat = "DOWN"".
TRIGGERED: GateClosed.
ACTIONEND.

ACTION TASK CarSafetyMeasure.
DECOMPOSITION:
  IF SemaStatIsGreen
    THEN StopTrain FI.
ACTIONEND.
```

In the action-object the construction STOP (*action*) is used to stop execution of the action *action*. For example, if TrainEnter executes LowerGateTimeout but at the same moment the gates close, GateIsDown stops the execution of LowerGateTimeout.
The EPOS-S constructions SET (*event*) and RESET (*event*) are used to enable and

disable events. Events are only enabled if the controller can expect one. For example, after execution of the action-object Lower the controller can expect the event GateClosed, so this event has to be enabled.
In these action-objects other actions like IncTrains, Lower, StopTrain and RestartTrain are used. The EPOS-S code of these actions can be found in Appendix A.

The condition that tests if the detected train is the first train, is written as condition-object NumTrainsIsOne. This object consists of code. The code that is used throughout the design of the application in EPOS-S, is the language C. Later, the tasks are written in a special language, based on the language C, that was specially designed for the simulator.

```
CONDITION NumTrainsIsOne.
CODE: "NumTrains == 1".
CONDITIONEND.
```

Besides the number of trains, the controller also has to know the number of cars on the crossing. The variable NumCars contains this number. Furthermore, it is also important that the controller knows the status of the gates and semaphore. These are stored in GateStatus and SemaStatus. These four variables have to be initialised. Therefore, the action module controller is changed in:

```
ACTION MODULE Controller.
DECOMPOSITION:
  Initialisation;
  WHILE TRUE
  DO Detect OD.
ACTIONEND.
```

During initialisation NumTrains and NumCars are set to zero, GateStatus becomes "UP" and SemaStatus "Green". The action Detect is now the infinite loop of waiting for events.

```
ACTION Detect.
DECOMPOSITION:
  PARALLEL
  ( WAIT FOR TrainEnters THEN TrainIn WAITEND,
    WAIT FOR TrainLeaves THEN TrainOut WAITEND,
    WAIT FOR GateClosed THEN GateIsDown WAITEND,
    WAIT FOR GateRaised THEN GateIsUp WAITEND,
    WAIT FOR CarEnters THEN CarIn WAITEND,
    WAIT FOR CarLeaves THEN CarOut WAITEND).
ACTIONEND.
```

We showed how the task TrainIn was developed. The actions TrainOut, CarIn and CarOut were designed similarly. The complete EPOS-S design can be found in Appendix A, including all event-, data-, condition- and interface-objects.
With EPOS-D, the document generator of EPOS, the system design in EPOS-S can be represented graphically. Figure 2.8 shows the flowchart of the action TrainIn. In Section 1.1 is explained that in the Event Table for all events that can occur, a list of

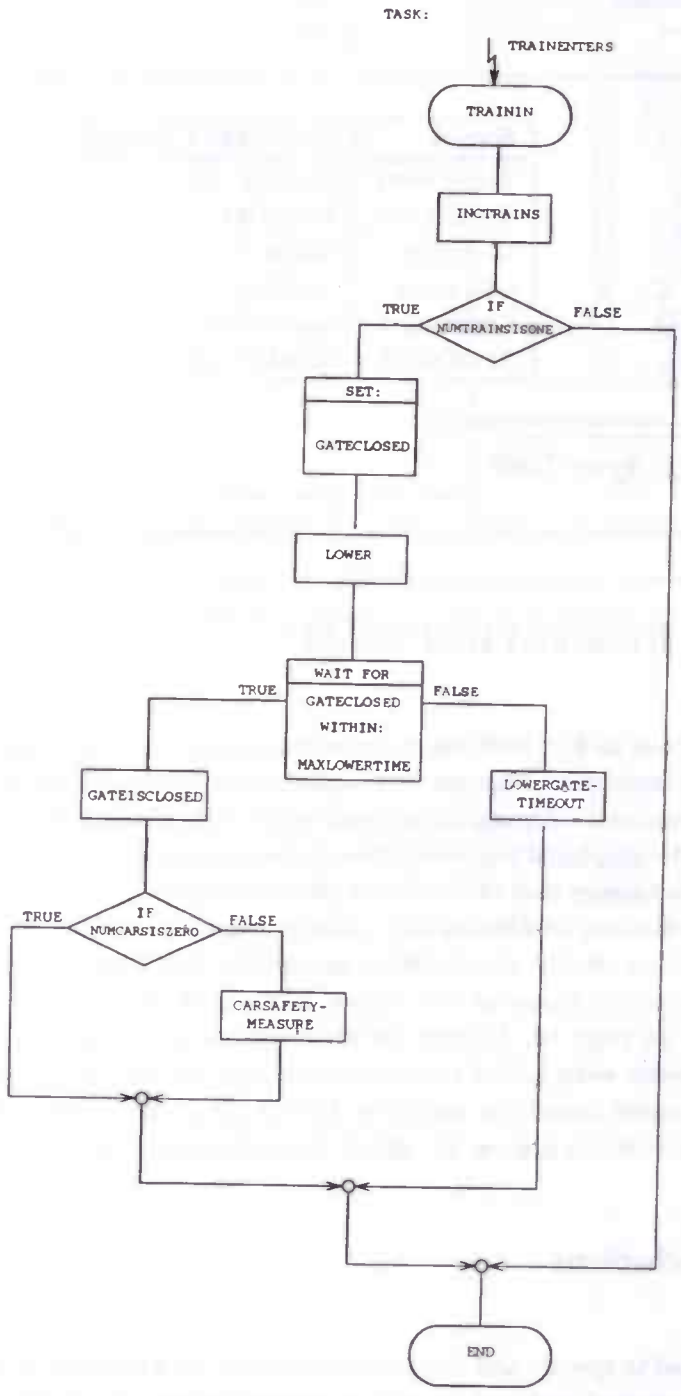


Figure 2.8: Flow Chart of the action-object *TrainIn*

associated tasks is given. Another result of the design of the application is, besides the tasks, that this table can be given. Table 2.1 shows the Event Table of the railroad crossing controller.

Event	**Associated tasks**
TrainEnters	*TrainIn*
TrainLeaves	*TrainOut*
CarEnters	*CarIn*
CarLeaves	*CarOut*
GateRaised	*GateIsUp*
GateClosed	*GateIsDown*

Table 2.1: Event Table

2.4 The front-end tool DA-FE

The specification in EPOS-S can be entered as text. But because graphical representations are often more adequate and easier to understand than text, EPOS provides graphical input modes through front-end tools. One of those graphical front-end tools is DA-FE. The graphical representation is automatically transformed to the EPOS-S specification language and entered into the database.
We designed the application of the railroad crossing with DA-FE. Figure 2.9 shows a DA-FE window during the development of the task *TrainIn*. This picture was created in the starting phase of the design. The final result of the design can be seen in Figure 2.8 on page 16. During the development of the application new constraints and requirements were added to the specification, so the system design also changed. In DA-FE is quite simple to realize in EPS-S the changes made in the specification. EPOS-S constructions are eas ily added, deleted or changed.

2.5 Conclusions

EPOS was used to specify and design the controller for a railroad crossing. Starting with a problem document and ending with a system design, the application was developed using stepwise refinement.

The application that we have developed consists of a set of tasks. These tasks are all small tasks. Small tasks will not occupy the task processor for long. We could also

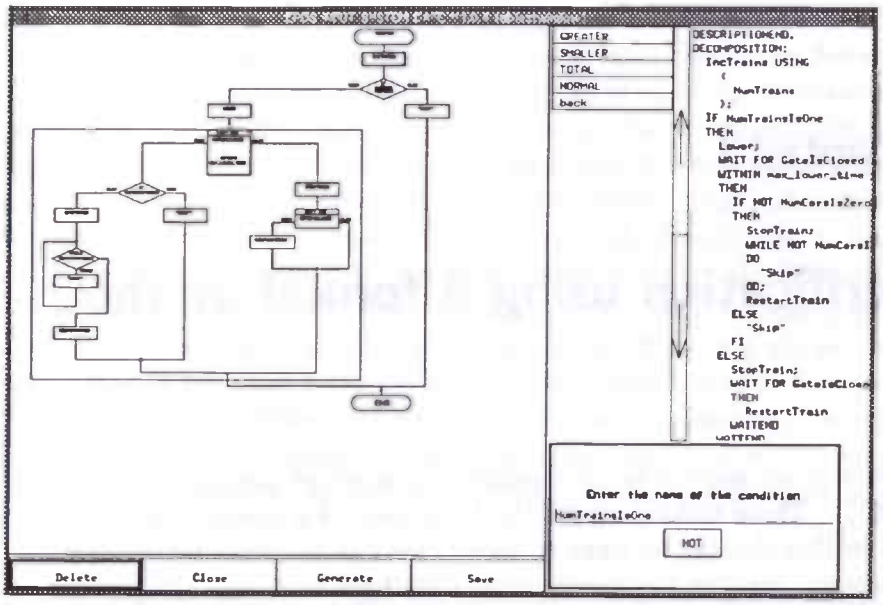


Figure 2.9: EPOS-S specification of the task *TrainIn* during the development

avoid that tasks are waiting for messages or events from the environment. The results is that a task will not hold the task processor occupied a long time. So, (critical) tasks do not have to wait a long time before they are assigned to the task processor.
An important requirement (see requirements 102 and 103 in Figure 2.4 on page 10) was that the railroad crossing should be safe. In EPOS-S these requirements are fulfilled in the tasks *TrainIn*, *LowerGateTimeout* and *CarSafetyMeasure*. In unsafe situations the gates are now lowered or the trains are stopped.

Chapter 3

Verification using a formal method

Generally an application is modelled as a set of tasks (for example in [Lau89] and [HS91]). These tasks describe the behaviour of a system. With the formal specification in this chapter we want to verify that the implementation of the system we have developed, satisfies the specification. We have to prove, using a formal method, that our implementation has the properties described in the Sections 1.2 and 2.2.1.
In literature several methods are used to specify a real-time application. In [GYF94] MASS is presented, a declarative language for specifying the reactive behaviour of real-time systems. A one-way railroad crossing was taken as example for the specification.
In [JM86] RTL (Real-Time Logic), a formal logic, is used to specify real-time systems. In this paper the safety analysis of timing properties in real-time systems are formalised. RTL is especially suitable for reasoning about timing behaviour of systems.
The specification of a real-time system in [BBvKT94] is done using the specification language VDM. The specification language is used to specify a software controller for a (toy) railroad system.
We have chosen to use Hoare-style logic to specify our application. Hoare-style logic is a simple logic that deals with functional requirements. The functionality of the tasks designed in EPOS-S is what has to be verified. We use wp-calculus, equivalent to Hoare-triples, to calculate a precondition, given the postcondition and the program code [DF84]. It is difficult to express pre- and postconditions for this application. Therefore, we use Statecharts to express predicates. With Statecharts it is easier to calculate the pre- and post conditions.
In the predicates the variable time will not be found, because the application handles the time aspects with events. For example, the time it takes to close the gates can be considered as the execution of the task *GateClosed* (which can be considered as the occurrence of the event TrainEnters) until the event GateIsDown occurs. The events are handled by the Statecharts.

In the first part of this chapter we discuss Statecharts. In the second part we specify each task using wp-calculus. After that, we determine for every task the task attributes and with these attributes we prove that the set of critical tasks is feasible.

3.1 Statecharts

Statecharts describe a system with the help of states and transitions between states. The controller for a railroad crossing can be considered as three separate systems: a system for trains, one for the gates and one for cars. The corresponding Statecharts are depicted in Figure 3.1 – 3.3 respectively. In Figure 3.1 the initial state is *None*. The variable that counts the number of trains in the region, NumTrains, is set to zero. When a train is detected, the state becomes *Approach*, NumTrains is increased by one and the signal 'lower' is sent to the gates. If more trains are detected the state remains Approach, but for every detected train NumTrains is increased by one. The event GateClosed causes the transition to another state: if no cars are on the crossing (NumCars = 0), the train can enter the crossing (the *Enter*-state) or else the train has to stop (the *Stop*-state). When the crossing is free again, the state becomes Enter. When a train has left the crossing, NumTrains is decreased and when the last trains has left the crossing, the state is changed to None and the signal 'raise' is sent to the gate.

The gate Statechart in Figure 3.2 has the initial state *Up*. When the signal 'lower', is received, the gates start moving down. As soon as they are down, the event GateClosed is generated. When the signal 'raise' is sent to the gates the new state becomes *MvUp* until the gate is open and the event GateRaised is received or until the 'lower' signal is received.

The car Statechart, as shown in Figure 3.3, is a very simple Statechart. It has two states: the initial state *None* and the state *Some*. Every time a cars enters the crossing, NumCars is increased and the state becomes or remains Some. If CarLeaves was generated NumCars is decreased and when the last car leaves the new state becomes None.

3.2 Verification using wp-calculus

With the Statecharts the status of the controller can now be described as the triple (Train, Gate, Car), with Train $\in$ {*None*, *Approach*, *Stop*, *Enter*}, Gate $\in$ {*Up*, *MvUp*, *MvDown*, *Down*} and Car $\in$ {*None*, *Some*}.
In the description of the application two requirements for the railroad crossing were made: firstly, the system has to be safe. In EPOS-R we have defined the term safe with two requirements (see Figure 2.4 on page 10). In terms of the Statechart we can say the crossing is safe if one of the following is true:

- No trains are approaching,
- A train approaches and the gates are moving down,
- A train enters the crossing, the crossing is empty and the gates are down,

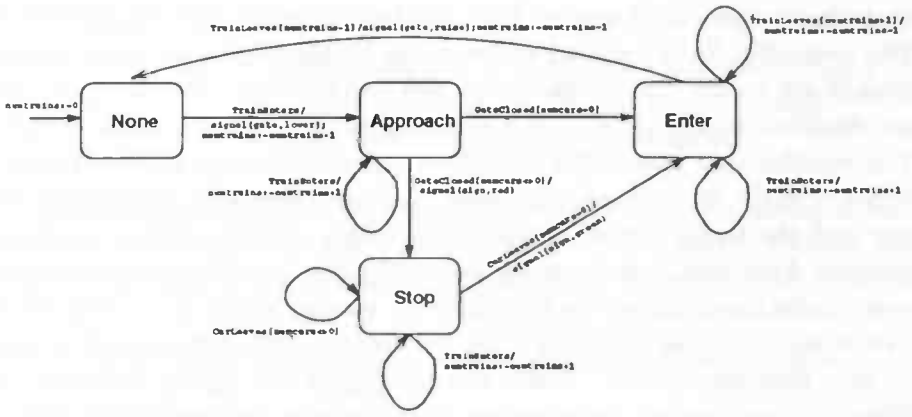


Figure 3.1: The train Statechart

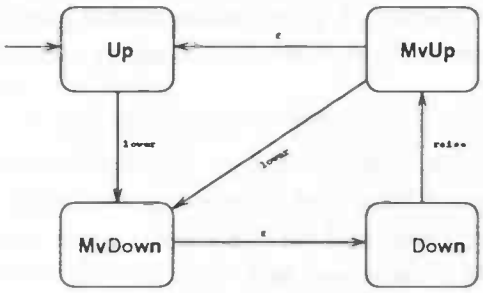


Figure 3.2: The gate Statechart

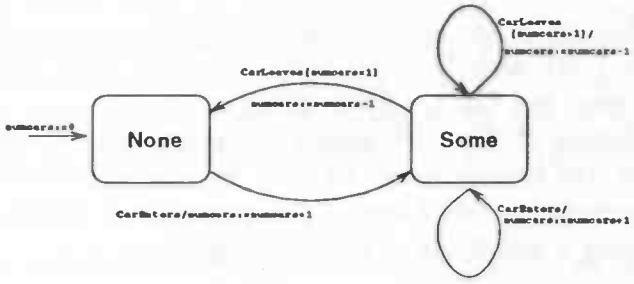


Figure 3.3: The car Statechart

- The trains are stopped.

This can be written as the following formula:

$$\begin{aligned}\textbf{Safe} \;=\; & (None, X_G, X_C) \vee (Approach,\ MvDown, X_C) \vee \\ & (Enter,\ Down,\ None) \vee (Stop, X_G, X_C)\end{aligned}$$

with $X_G \in \{Up,\ MvDown,\ Down,\ MvUp\}$ and $X_C \in \{None,\ Some\}$.

Secondly, cars should not wait unnecessarily long. This is called the progress requirement. In terms of the Statecharts one of the following must be satisfied:

- No trains are approaching and the gates are up or moving up,
- A train approaches,
- A train enters the crossing,
- The trains are stopped, the gates are down and one or more cars are on the crossing.

Maybe the last point is difficult to understand. It guarantees that a train will not stop without a reason. Only when the gates are down and one ore more cars are on the crossing the train will stop.
This results in the following formula:

$$\begin{aligned}\textbf{Progress} \;=\; & (None,\ MvUp, X_C) \vee (None,\ Up, X_C) \vee \\ & (Approach, X_G, X_C) \vee (Stop,\ Down,\ Some) \vee \\ & (Enter, X_G, X_C)\end{aligned}$$

with $X_G \in \{Up,\ MvDown,\ Down,\ MvUp\}$ and $X_C \in \{None,\ Some\}$.

The railroad crossing controller must guarantee that the system is always safe and that there will be progress. Therefore, the invariant reads as follows: **Safe** $\wedge$ **Progress**. We find, after some calculation, the invariant **I**

$$\begin{aligned}\mathbf{I} \;=\; & (None,\ MvUp, X_C) \vee (None,\ Up, X_C) \vee (Approach,\ MvDown, X_C) \vee \\ & (Enter,\ Down,\ None) \vee (Stop,\ Down,\ Some)\end{aligned}$$

with $X_G \in \{Up,\ MvDown,\ Down,\ MvUp\}$ and $X_C \in \{None,\ Some\}$.

From the description of the railroad crossing (see Section 1.2) we know what should happen when an approaching train is detected. The Statecharts in the Figures 3.1

to 3.3 show the situation when a train is detected: NumTrains is increased and, if necessary, the gate closes. Now we can derive the postcondition, using the terms from the Statecharts.

$$\begin{aligned} \text{Post}_{TrainIn} &= \mathbf{NumTrains} \geq 1 \wedge ((Approach, MvDown, X_C) \vee \\ &\quad (Stop, Down, Some) \vee (Enter, Down, None)) \\ &\Rightarrow \mathbf{NumTrains} \geq 1 \wedge \mathbf{Safe} \wedge \mathbf{Progress} \end{aligned}$$

The task *TrainIn*, as depicted in Figure 2.8 on page 16, has to be translated into code so that task can be executed on the task processor. The code of *TrainIn* is shown in Figure 3.4. The code is a translation from EPOS-S language into code used by

```
TASK TrainIn
  numtrains = numtrains + 1;
  IF numtrains == 1 THEN
    ENABLE(gate_closed);
    SIGNAL(gate, lower);
    IF gatestatus == mvup THEN
      REMOVE(RaiseGateTimeout)
    FI;
    gatestatus = mvdown;
    INSERT(LowerGateTimeout) AFTER max_lower_time;
    IF numcars != 0 THEN
      INSERT (CarSafetyMeasure);
    FI;
END; /* TrainIn */
```

Figure 3.4: The code for the task *TrainIn*

the simulator (see Chapter 4). The execution of an EPOS-S object corresponds with `INSERT(`*task*`)`, which means that the task is inserted into the Ready Queue of the scheduler. If this statement is followed by `AFTER(`*time*`)`, the scheduler waits a certain time before the task is inserted, this is a translation of the EPOS-S construction `WITHIN` *time*. The statement `REMOVE(`*task*`)` tells the scheduler to remove the task from the Ready Queue or from the Time Table, in EPOS-S the `STOP` construction is used for this.

Considering the postcondition and the code of the task *TrainIn*, we can determine the precondition.

$$\begin{aligned} wp\,(TrainIn, Post) &= wp\,(TrainIn, \mathbf{NumTrains} \geq 1 \wedge \\ &\quad ((Approach,\ MvDown,\ X_C) \vee (Stop,\ Down,\ Some) \vee \\ &\quad (Enter,\ Down,\ None))) \end{aligned}$$

$$
\begin{aligned}
&\Rightarrow\ wp(S_1, (\mathbf{NumTrains} = 1 \wedge (None,\ Up,\ X_C)) \vee \\
&\quad (\mathbf{NumTrains} > 1 \wedge ((Approach,\ MvDown,\ X_C) \vee \\
&\quad (Stop,\ Down,\ Some) \vee (Enter,\ Down,\ None)))) \\
&\equiv\ (\mathbf{NumTrains} = 0 \wedge (None,\ Up,\ X_C)) \vee \\
&\quad (\mathbf{NumTrains} > 0 \wedge ((Approach,\ MvDown,\ X_C) \vee \\
&\quad (Stop,\ Down,\ Some) \vee (Enter,\ Down,\ None))) \\
&\equiv\ \mathbf{NumTrains} \geq 0 \wedge ((None, Up, X_C) \vee \\
&\quad (Approach,\ MvDown,\ X_C) \vee \\
&\quad (Stop,\ Down,\ Some) \vee (Enter,\ Down,\ None)) \\
&\Rightarrow\ \mathbf{NumTrains} \geq 0 \wedge \mathbf{I}
\end{aligned}
$$

Similarly, we can formulate the postconditions of the other tasks. With these and the code of the tasks we can calculate the preconditions. The code of every task can be found in Appendix B. We now give the pre- and postcondition for every task.

$$
\begin{aligned}
\mathrm{Pre}_{TrainOut} &= (\mathbf{NumTrains} > 0 \wedge (Enter,\ Down,\ None)) \\
\mathrm{Post}_{TrainOut} &= (\mathbf{NumTrains} = 0 \wedge (None,\ MvUp,\ X_C)) \vee \\
&\quad (\mathbf{NumTrains} > 0 \wedge (Enter,\ Down,\ None))
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Pre}_{CarIn} &= \mathbf{NumCars} \geq 0 \wedge (X_T,\ \neg Down, X_C) \\
\mathrm{Post}_{CarIn} &= \mathbf{NumCars} > 0 \wedge (X_T,\ \neg Down,\ Some)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Pre}_{CarOut} &= (\mathbf{NumCars} > 0 \wedge ((\neg Enter,\ \neg Down,\ None) \vee \\
&\quad (\mathbf{NumCars} > 0 \wedge (Enter,\ Down,\ None) \vee \\
&\quad (Stop,\ Down,\ Some)) \\
\mathrm{Post}_{CarOut} &= (\mathbf{NumCars} = 0 \wedge (Enter,\ Down,\ None)) \vee \\
&\quad (\mathbf{NumCars} > 0 \wedge (Enter,\ Down,\ None) \vee \\
&\quad (Stop,\ Down,\ Some))
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Pre}_{GateIsDown} &= \mathbf{NumTrains} \geq 1 \wedge (Approach,\ MvDown,\ X_C) \\
\mathrm{Post}_{GateIsDown} &= \mathbf{NumTrains} \geq 1 \wedge \\
&\quad ((Enter,\ Down,\ None) \vee (Stop,\ Down,\ Some))
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Pre}_{LowerGateTimeout} &= \mathbf{NumTrains} \geq 1 \wedge (Approach,\ MvDown,\ X_C) \\
\mathrm{Post}_{LowerGateTimeout} &= \mathbf{NumTrains} \geq 1 \wedge (Stop,\ MvDown,\ X_G, X_C)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Pre}_{GateIsUp} &= \mathbf{NumTrains} = 0 \wedge (None,\ MvUp,\ X_C) \\
\mathrm{Post}_{GateIsUp} &= \mathbf{NumTrains} = 0 \wedge (None,\ Up,\ X_C)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Pre}_{RaiseGateTimeout} &= \mathbf{NumTrains} = 0 \wedge (None,\ \neg Up,\ X_C) \\
\mathrm{Post}_{RaiseGateTimeout} &= \mathbf{NumTrains} = 0 \wedge (None,\ \neg Up,\ X_C)
\end{aligned}
$$

$$\begin{aligned} \text{Pre}_{CarSafetyMeasure} &= \mathbf{NumTrains} \geq 1 \wedge \mathbf{NumCars} \geq 1 \wedge \\ &\quad (Approach,\ Down,\ Some) \\ \text{Post}_{CarSafetyMeasure} &= \mathbf{NumTrains} \geq 1 \wedge \mathbf{NumCars} \geq 1 \wedge \\ &\quad (Stop,\ Down,\ Some) \end{aligned}$$

After some calculation we can see that the preconditions of the tasks are weaker than the invariant, so we can conclude that **I** holds for every task.

3.3 Task attributes

Recall from Section 1.1 that every task has five attributes (release time, start time, deadline, worst-case execution time and criticalness). In the next sections we explain how we determined the deadline, worst-case execution time and criticalness of the tasks.

3.3.1 Criticalness

If the violation of a task's deadline leads to an unsafe situation, we call this a critical task. Four tasks in our task set can be considered critical: *TrainIn*, *CarIn*, *LowerGate-Timeout* and *CarSafetyMeasure*.
If one of these tasks misses its deadline, the situation can be become dangerous. For example, if a train is the only train in the critical region, and *TrainIn* has not finished execution before its deadline, the gates will not lower in time. This can have the result that the train will enter the crossing while cars are still using it. An accident is possible and lives of many people may be in danger.
If a non-critical task misses its deadline, we do not have to be afraid that the situation becomes dangerous. However, the status of the controller is not correct anymore, for instance, the variable NumTrains can contain the wrong number of trains, if *TrainOut* has missed its deadline. To put it right, the system has to be reset, for example by the traffic controller of the railroad company.

3.3.2 Deadlines

Assume a train approaches the crossing with a maximum speed of 160 km/h. When a train is detected 3 km before the crossing, it takes at least 67.5 seconds before the train arrives. If lowering the gate takes maximal 30 seconds and a trains needs 1 km to stop (i.e. 22.5 seconds at top speed), there is a margin of

$$67.5 - (30 + 22.5) = 15 \text{ seconds}$$

This is the deadline of the task *TrainIn*.

The deadline for *TrainOut* is determined as follows. A third train can enter the crossing if the first train has left. The time between two trains going in the same direction is at least three minutes (see Section 2.2.1). So, the execution of *TrainOut* from the first train has to be finished at most three minutes after he has left. The deadline of *TrainOut* may not be larger than 180 seconds. We have chosen for a deadline of 60 seconds, because three minutes is very long compared to the execution times of the tasks.
The crossing that we have taken as example for our application (see Section 5.1.2) is situated in a town. In a town the speed limit for cars is 50 km/h, but there will always be car-driver who drive too fast. Therefore we assume that cars do not drive faster than 80 km/h. The two sensors (CarEnters and CarLeaves) have a mutual distance of 50 metres. So, a car driving 80 km/h leaves the crossing in 2.2 seconds. The task *CarIn* has to be finished before the car has left the crossing, the deadline of *CarIn* may not be larger than 2.2 seconds.
The deadline of *CarOut* is set at 7.5 seconds, because it always takes about 10 seconds before a 4^{th} car enters the crossing after the first has left it (note, there is a maximum of six car that can be on the crossing at the same time, so only three can be at one half of the crossing).
Two very important tasks are *LowerGateTimeout* and *CarSafetyMeasure*, which have to be executed as soon as possible. So, their deadlines must be as small as possible. Assume the task with the largest worst-case execution time, **max**.e say, is just assigned to the task processor. It takes at most **max**.e ms before the processor is available, since we do not have pre-emption. The deadlines of both tasks are now LowerGateTimeout.e + **max**.e and CarSafetyMeasure.e + **max**.e. As soon as we know the execution times of every task, we can calculate the deadlines of these two tasks.
The deadlines of the other tasks are not as important as it is for critical tasks, therefore we have given them large values.

3.3.3 Worst-case execution times

The worst-case execution times (wcet) of the tasks are calculated using the tasks' code and the model of the task processor. Every statement takes an amount of time to be executed. The worst-case execution times are calculated on the basis of these costs. A fixed amount of time is included for starting up a task and terminating a task. The user of the simulator is able to change these values (see Chapter 4).

This results in the following values for the attributes deadline and critical. Note that the deadlines for the tasks *LowerGateTimeout* and *CarSafetyMeasure* are not known yet.

Task	Deadline (in ms)	Critical
TrainIn	15000	yes
TrainOut	60000	no
CarIn	2200	yes
CarOut	7500	no
GateIsDown	7000	no
LowerGateTimeout	unknown (yet)	yes
GateIsUp	7000	no
RaiseGateTimeout	4000	no
CarSafetyMeasure	unknown (yet)	yes

Table 3.1: A part of the Task Table

3.4 Feasibility proof

It is the responsibility of the application programmer to prove that the set of critical tasks is feasible. In [Tol95] a method is given as an aid for this proof. We will use this method and prove that our set of critical tasks is feasible.

Suppose, C is a subset of B, the set of critical tasks. We can calculate the load fraction of this subset as follows. If the tasks in C have execution times that can be bound by a number ee, the relative deadlines have a lower bound dd, the tasks in C are at least p time released after each other, and $C = \{b_j \mid j \in \mathbf{N}\}$ satisfies

$$\begin{aligned} & (\forall\, j : b_j.r \leq b_{j+1}.r \ \wedge\ b_j.d \leq b_{j+1}.d) \\ \wedge\ & p > 0 \ \wedge\ (\forall\, j,k : j \leq k : b_k.d - b_j.r \geq (k-j)\cdot p + dd) \\ \wedge\ & 0 < ee < dd - ub \ \wedge\ (\forall j :: b_j.e \leq ee) \end{aligned} \tag{3.1}$$

the the load fraction u can be defined as:

$$u = \frac{ee}{\mathbf{min}\,(p, dd - ub)}$$

We assume that the task processor can be made available for critical tasks in time $\leq at$. The constant ub is an upper bound of both at and the execution times of the critical tasks, so ub satisfies:

$$at \leq ub \ \wedge\ (\forall\, b \in B :: 0 \leq b.e \leq ub)$$

For the proof of feasibility we have to choose partitions $\{B_i\}_i$ of the critical task set B, to calculate the load fractions of these partitions such that

$$u_i \geq 0, \text{ with } \sum_i u_i \leq 1 \tag{3.2}$$

We have to determine for each B_i constants ee, p and dd satisfying Formula 3.1.

The critical task set for the application of the railroad crossing is {*TrainIn, CarIn, LowerGateTimeout, CarSafetyMeasure*}. We choose $\{TrainIn_i\}$ as one partition of B. Assume that trains going in the same direction are separated by at least three minutes and that the execution times can be bounded by max.e ms ($= ub$). We take $ee =$ *TrainIn*.e ms. The lower bound of the deadlines (dd) is 15000 ms and the period p is set to 180000 ms, the time that separates two trains going in the same direction. Because we have two tracks, we have to multiply $u_{TrainIn}$ by 2. We can now calculate the load fraction u of this subset

$$\begin{aligned} u_{TrainIn} &= 2 \cdot (TrainIn.\text{e}/\mathbf{min}\,(180000, 15000 - ub)) \\ &= 2 \cdot (TrainIn.\text{e}/(15000 - ub)) \\ u_{TrainIn} &\geq 0, \text{ if } ub < 15000 \end{aligned}$$

This partition satisfies Formula 3.1 under the given condition.

We know that only three cars can be at the same half of the crossing at the same time. When the cars drive fast, after 2.2 seconds the cars have left and three other cars can be present. So, if we take $\{CarIn_i\}$ as the next partition of B, the period p is equal to $2200/3 \approx 735$ ms, $ee =$ *CarIn*.e ms and $dd = 2200$ ms. The load fraction, taking into account the fact that cars drive in two directions, of this partition is

$$\begin{aligned} u_{CarIn} &= 2 \cdot (CarIn.\text{e}/\mathbf{min}\,(735, 2200 - ub)) \\ u_{CarIn} &\geq 0, \text{ if } ub < 2200 \end{aligned}$$

With the choice of these values for ee, p and dd, Formula 3.1 holds under the given condition.

The two other task that are in the critical task set, are only executed in exceptional situations. After this, the system has to be reset by an operator, until that is done, no other tasks will be executed. Therefore, these two task do not influence the feasibility of the task set.

The load fraction of the task set {*TrainIn, CarIn, LowerGateTimeout, CarSafety-Measure*} is

$$\begin{aligned} u &= u_{TrainIn} + u_{CarIn} \\ &= (2 \cdot TrainIn.\text{e})/(15000 - ub) + (2 \cdot CarIn.\text{e})/\mathbf{min}\,(735, 2200 - ub)) \quad (3.3) \\ &\quad \text{, if } ub < 2200 \end{aligned}$$

We can now conclude that, following Formula 3.2, the set of critical tasks is feasible if the upper bound ub is less than 2200 and the sum of Formula 3.3 is less than 1. In Chapter 5 we calculate the worst-case execution times of the tasks. Then we can determine if the set of critical tasks is feasible.

3.5 Conclusions

We have used wp-calculus method to verify the design of the controller of the railroad crossing. To be able to formulate pre- and postconditions as well as an invariant, we used Statecharts to describe the controller of the railroad crossing. With the specification given in the Sections 1.2 and 2.2.1 we were able to give an invariant of the system. We proved that the invariant remains invariant when the developed tasks are executed. We can conclude that our controller satisfies the specification.
One of the requirements of ARTIE for the application is that the set of critical tasks has to be feasible. Therefore, we first had to assign attributes to the tasks. With these attributes we proved that the set of critical tasks is feasible if $ub < 2200$ and the sum of Formula 3.3 < 1. We will show this in Chapter 5 when the execution times of the tasks are known. We can conclude that no critical task will miss its deadline if the given conditions are satisfied.

Chapter 4

Implementation of the simulator

The application, the railroad crossing controller, is meant to be a test case for ARTIE. This can only be the case if the application is executed on the architecture, and therefore, we have to implement ARTIE. To be able to test ARTIE we also need to simulate the environment of the real-time application. The simulator consists of the following parts:

- An implementation of ARTIE.
  The following two parts are implemented:
  - The kernel processor, running the kernel programs
  - The task processor, running the application
- A simulator of the railroad crossing's environment

The implementation of the kernel and the task processor are independent from the application. But the environment and the set of tasks differ per application. In Appendix C.2 is explained how the application of the railroad crossing controller can be replaced by another application and how the environment has to be adjusted.

To evaluate the architecture we need information about its behaviour. Therefore, some features are added to ARTIE to collect the needed information. In Chapter 5 we will explain which data are gathered and how these data can be used to evaluate ARTIE.

The three parts of the simulator need to exchange data, so communication is important. In the next Section the communication protocol for the information exchange between the kernel, the task processor and the environment is discussed.

4.1 The communication protocol

The description of ARTIE requires that the kernel runs its own processor. Because the simulator has to run on a UNIX machine as well as on a MS-DOS machine with a single kernel processor connected to it, we can only use one bi-directional communication channel. Therefore, the task processor and the environment are merged into one part. We call this part of the simulator the controller.
In Figure 4.1 the setup of the simulator is depicted. The thin vertical dark-gray bar represents the communication channel between the controller and the kernel. In this picture we can also see the information streams between the kernel, the task processor and the environment. The information streams from the task processor and the environment to the kernel are merged to create one stream. To guarantee good com-

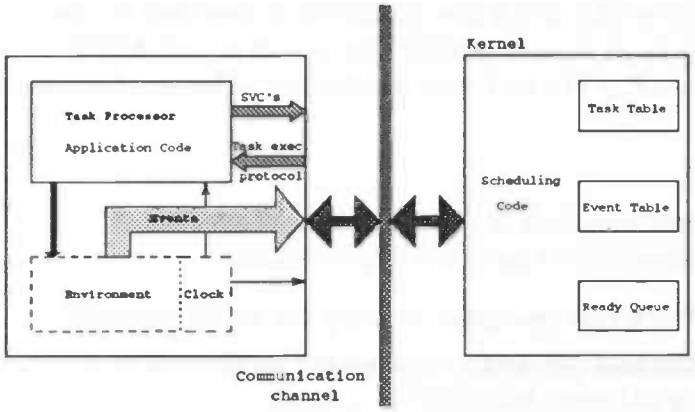


Figure 4.1: The three parts of the simulator

munication between the kernel, the task processor and the environment, we need a communication protocol. We also need a protocol to steer the simulation. For example, we need commands to terminate the simulation, as well as commands to generate debugging output. At the start of a simulation the protocol prescribes that the kernel receives the Task Table and the Event Table.
Finally, the external events also belong to the communication protocol. When an event occurs in the environment, a message containing the event-identifier and the time of occurrence is sent to the kernel.
We distinguish two separate message streams. One flows from the controller to the kernel and the other in the reverse direction. Both programs, the controller and the kernel, consist of a loop waiting for messages sent over the channel. Special care has to be taken to avoid that both sides of the channel are waiting for a message, at the same time.

4.2 Implementation of ARTIE

4.2.1 The kernel

In this part we explain how the kernel programs of ARTIE are implemented. When the Event Table and the Task Table are received by the kernel, it waits until a message arrives.
If an event occurs, the kernel retrieves the associated tasks from the Event Table. The tasks are put into the set of ready tasks, or as we call it, the Ready Queue. The values of the attributes of the tasks, needed for scheduling the tasks, are retrieved from the Task Table. The kernel gives as output the set of ready tasks every time the set is changed. Figure 4.2 shows the Ready Queue during a simulation run. The RT stands for 'running task', the task assigned to the task processor. Tasks that have to wait before they are scheduled into the Ready Queue, are stored in the Time Table. At sched_time they are inserted into the Ready Queue. The kernel may also receive messages to insert a task

```
-----[ BGN READY QUEUE << 003546 >> ] -----
   nr | startt | rel.t | wcet | deadl | crt | taskid
   ---+--------+-------+------+-------+-----+-------
RT: 0 |   3546 |  3321 |  240 |  3967 |  Y  | 1000
    1 |        |  3438 |  125 |  6893 |  N  | 1001

    ======= BGN Time Table =======
     sched_time | taskid | deadline
     -----------+--------+---------
         7001   |  1006  |     311
         7451   |  1008  |     154
    ======= END Time Table =======

-----[ END READY QUEUE ] -----
```

Figure 4.2: The Ready Queue and Time Table: output from the kernel during a simulation

into, or to remove a task from the Ready Queue. We call these messages supervisor calls (SVC's). When a task has to be inserted, this is done according the earliest-critical-deadline-first strategy.
When the task processor becomes idle, the kernel receives a message to send the next task in the schedule to the task processor, if one is available. The running task is removed from the Ready Queue and the next task in the schedule is made running. If no tasks are available in the Ready Queue the simulated time is increased until an event occurs.

4.2.2 The task processor

The model of the task processor consists of the tasks code and a protocol for communication with the kernel and environment. If the processor is idle, a message is sent to the kernel with the request for the next task that has to be made running. If the task processor has received the task, it will execute the task's code. The execution time of a task depends on the number and the type of instructions and on the model of the task processor. To calculate the execution times of the tasks, the user is allowed to determine the time costs per instruction type. The tasks code contains the following instruction types:

- Assignment
- Increment
- Decrement
- if-statement
- Equality test
- Sending a message
- Sending a signal

Costs for starting and terminating a task are added to the execution time. During simulation, for every task its execution time is calculated using the given time costs.

The task that is running on the task processor can also request the kernel to insert a task into, or remove a task from the Ready Queue.

If the application programmer wants to replace the application by another one, he has to replace the code of the tasks plus the procedures that are used to calculate the execution times. In Appendix C.2 we give a full description how an application can be replaced.

4.3 Simulation of the environment

The simulation of the railroad crossing's environment is discussed in this part. Before the simulation starts, the environment reads the input data given by the user. This input data consist of events and the points in time when the events should occur. The environment generates the event when the simulator reaches that point in time. When a change in the environment takes place, the simulator generates output. In Figure 4.3 the output of the environment at $t = 0$ is given. The `Event List` contains

```
----[ BEGIN ENVIRONMENT STATE (000000) ]----
Number of trains :   0
Number of cars   :   0
Gate status      : UP
Semaphore status : GREEN

Event List:  (<2000, TrainEnters>,
              ...,
              <90000, StopSim>)
----[ END ENVIRONMENT STATE ]----
```

Figure 4.3: Output from the environment

the input data from the user. <2000, TrainEnters> stands for the occurrence of event TrainEnters at $t = 2000$.
This part of the simulator also contains the simulator's clock. Events are generated and tasks are scheduled with respect to this clock.

The user has also to define application specific constants, like how long it may take for the gates to lower and raise. In Section 5.1.2 these constants are defined for our simulation sessions.

In Table 4.1 we give the tasks and their identification numbers as used in the simulator.

Task	**taskid**		**Event**	**eventid**
TrainIn	1000		TrainEnters	2002
TrainOut	1001		TrainLeaves	2003
CarIn	1002		CarEnters	2004
CarOut	1003		CarLeaves	2005
GateIsDown	1004		GateClosed	2000
LowerGateTimeout	1006		GateRaised	2001
GateIsUp	1005			
RaiseGateTimeout	1007			
CarSafetyMeasure	1008			

Table 4.1: Task - and event identifiers

Chapter 5

Simulation and results

Now that we know how the simulator of the railroad crossing is implemented, we finally are able to simulate the railroad crossing. During the simulation runs, we retrieve information about the kernel and the task processor. With the results it is possible to draw conclusions about ARTIE's behaviour.
In the following sections the goals of the simulation sessions are determined. The precise setup of the simulation session is given and we discuss how information from the kernel is retrieved and how it has to be interpreted. Finally, the results are presented and discussed.

5.1 Simulation of the railroad crossing

5.1.1 Collecting information

We only can start simulating if we know why we do it. It is not the purpose to simulate the application and see if it was designed correctly. It is more interesting to examine how ARTIE behaves under different circumstances. By collecting the right data during the simulation sessions, it is possible to get the information we want. With the results of the simulation runs conclusions can be drawn with regard to the behaviour of the architecture.

CPU utilisation often gives a good indication about the performance of a system. In [LL73] and [Gom93] the CPU utilisation of periodic tasks is defined as:

$$\frac{C_i}{T_i}$$

where C_i is the worst-case execution time and T_i is the period of task t_i.
For aperiodic tasks it is assumed that they will arrive with a certain period. The CPU

utilisation is defined similarly. The total CPU utilisation of a task set is the sum of the utilisations of each task.
We don't use this definition for CPU utilisation, because in our application we don't have cyclic tasks, and from the aperiodic tasks we don't always know the period in which they arrive. For example, *CarSafetyMeasure* is a task that is only executed in an exceptional situation. We do not know how often this situation will occur. Another reason for not using this definition is that cars can't enter the crossing during a long time when the gates are down. So, the sum of the utilisations for each task does not give a good indication for the CPU utilisation of our task set. For our simulation we define the CPU utilisation as follows: the amount of time the processor is running divided by the total simulation time. After each simulation session we know how many percent of the time during simulation the processor was busy.

In the literature several definitions can be found for the response time. In [Hen89] it is defined as the (absolute) deadline of a task minus the current time. It indicates how much time is left before a task reaches its deadline.
We are interested in how long it takes for a task to respond to the request to be executed. We take the time between the release of a task and its termination, and call this the response time. In terms of task attributes the response time of task b is defined as $b.s + b.x - b.r$, with $b.x$ the actual execution time of task b.

Finally, we want to know how 'full' the Ready Queue is during simulation. We call the Ready Queue 'full' if the schedule is not feasible, e.g. there is a task that is not able to meet its deadline. The i^{th} task in the Ready Queue (b_i) has a starttime not later than

$$s(i) = b_0.s + \left(\sum_{j=0}^{j=i-1} b_j.e \right),$$

where b_0 is the running task.
In [Tol95] the load of the Ready Queue is now defined as

$$\max \left(i : 0 \leq i < N : \frac{b_i.e}{b_i.d - s(b_i)} \right) \tag{5.1}$$

with N the number of tasks in the Ready Queue.
Every time the Ready Queue changes during simulation, the load is updated.

5.1.2 Setup of the simulations

To make the simulation more realistic, we take an existing railroad crossing as example. On the railroad crossing near the station of Haren, on the route Zwolle – Groningen, each hour three trains travel from Groningen into the direction of Zwolle and three trains go in the reverse direction. In a very busy period, eight goods trains can also

use the crossing in an hour. So, in the period of one hour maximal fourteen trains use the railroad crossing. Furthermore, an average of eighty cars cross the crossing in one hour.

In Section 4.2.2 a number of variables that have an effect on the environment are defined and explained. For the various simulation sessions these variables can be changed. In Table 5.1 we give the values of the costs for each instruction. With these values and the code of the tasks (see Appendix B) the worst-case execution times can be determined. During simulation these values are used to calculate the actual execution times of the tasks. We have taken rather large values to simulate a slow processor. If the results of the tests are satisfying in this case, a faster processor will certainly satisfy. For

Statement	Costs (ms)
assignment	8
decrement	2
increment	2
if evaluation	8
equality test	6
sending a message	24
sending a signal	16
starting a tasks	80
terminating a task	16

Table 5.1: Costs of the statements

the task *TrainIn* we give the calculation of its worst-case execution time. Consider the code of the task as it is depicted in Figure 3.4 on page 23. For each instruction we take the associated value as given in Table 5.1. Note that the instructions `INSERT`, `REMOVE`, `ENABLE` and `DISABLE` are equivalent with sending a message. The evaluation of the `IF`-statement and both alternatives is a special case. The statement itself takes 8 ms to evaluate. The `THEN`- and `ELSE` cost not more than the maximum of both. We finally get the following addition to determine the worst-case execution time of *TrainIn*:

$$80 + 2 + 8 + \mathbf{max}\,(24 + 16 + 8 + \mathbf{max}\,(24, 0) + 24 + 24, 0) + 16 = 240$$

The worst-case execution time for every task is given in Table 5.2. In Section 3.3.2 we were not able to find the deadlines of the tasks *LowerGateTimeout* and *CarSafetyMeasure*, because the execution times of these tasks were unknown. Now that we know these execution times we can determine the deadlines of these two tasks. We see that the maximum execution time is 240 ms. The deadline of *LowerGateTimeout* is now $150 + 240 \approx 400$ and *CarSafetyMeasure* has a deadline of $126 + 240 \approx 370$.

Task	Deadlines	WCET	Critical
TrainIn	15000	240	yes
TrainOut	60000	184	no
CarIn	2200	142	yes
CarOut	7500	164	no
GateIsDown	70000	184	no
LowerGateTimeout	400	150	yes
GateIsUp	7000	152	no
RaiseGateTimeout	4000	120	no
CarSafetyMeasure	370	126	yes

Table 5.2: The complete Task Table

Now we can also determine if the set of critical tasks is feasible using Formula 3.3 on page 28. Now it is possible to fill in the missing values. The upper bound of the execution times (ub) is 240 ms $<$ 2200, *TrainIn*.e is 240 ms and *CarIn*.e is 142 ms. This results in the following formula:

$$\begin{aligned} u &= (2 \cdot \mathit{TrainIn}.\mathrm{e})/(15000 - ub) + (2 \cdot \mathit{CarIn}.\mathrm{e})/\mathbf{min}\,(735, 2200 - ub)) \\ &= (2 \cdot 240)/(15000 - 240) + (2 \cdot 142)/735 \\ &= 0.0325 + 0.3864 \\ &= 0.4189 \end{aligned}$$

Thus $u \leq 1$ and we can conclude that the set of critical tasks is feasible.

We also have to define some time limits for the simulation, for example, a train can not be infinitely long in the critical region, and the gate has to be closed once. Therefore, we define the following boundaries, with t in seconds.

The time a train spends in the critical region	$75 \leq t \leq 150$
The time a car spends on the crossing	$2.2 \leq t \leq 6$
Time to lower the gates	$15 \leq t \leq 30$
Time to raise the gates	$20 \leq t \leq 35$
Time for a car to leave the crossing after the gates have closed	$t = 7$

With various simulation runs we try to examine different aspects of ARTIE. In the first place we want to test ARTIE in the normal situation. By far, most of the time nothing special happens on a railroad crossing. We also try out the limits of the system. When is the CPU utilisation extra high? In which situations is the Ready Queue not feasible anymore? Questions like these we try to answer with the simulations.
Finally, we simulate exceptional situations like a car crashing through a gate, a car stuck

on the crossing and a situation in which the gates never open again. It is important that ARTIE also behaves well in these situations.

5.1.3 Expectations of the experiments

In a normal situation we expect that the load of the Ready Queue will not get very high, because the deadlines of the tasks are much larger than the worst-case execution times. So, often there will be enough slack. During the simulation run, we expect that the load will be zero most of the time, because e.g., when a train is spotted and the gates are closed, no other tasks than *TrainOut* and another *TrainIn* can be executed. We think that in normal situations the task processor often is idle. So, the CPU utilisation will be low.
Because the time between two events is rather long in normal situations, the expectation is that tasks have often finished execution before a new event has occurred. The response time of the tasks will often be the execution time of the task.

We expect that overload situations do not occur because we proved that the set of critical tasks is feasible. This is proven with the assumption of a two track railroad. When more than two trains and/or more than two cars enter the crossing at the same time, the Ready Queue has not to be feasible anymore.

From the definition of the CPU utilisation (see Formula 5.1.1 on page 35), we can expect that when cars enter the crossing close behind each other with high speed, the utilisation will be high. The (worst-case) execution time of *CarIn* is 142 ms and *CarOut* 164 ms. The period between two cars is taken 400 ms. In this case, the total CPU utilisation is expected to be, $0.765 (= 142/400 + 164/400)$.

In the other exceptional cases, we do not expect that the load or CPU utilisation are very different from the normal situation. Compared to the normal situation, the number of tasks executed is equal or even less in these situations.

5.2 Results

5.2.1 A normal situation

The input data for the simulation of a normal situation consist of the events TrainEnters and CarEnters. We take the crossing in Haren (see Section 5.1.2) as example. Two cars enter the crossing followed by two trains. When the trains have left the crossing five cars, that have been waiting, enter close behind each other. After a while, three cars cross over followed by a train. When the gates are open again, four cars use the crossing and when they have left, the simulation terminates. The exact input data are given in Figure 5.1. This simulation takes one million clockticks, what a representation is of

sixteen minutes and forty seconds (one clock-tick corresponds to 1 ms).

```
(<010000, CarEnters>, <020000, CarEnters>,
 <040000, TrainEnters>, <050000, TrainEnters>,
 <260000, CarEnters>, <270000, CarEnters>,
 <280000, CarEnters>,  <290000, CarEnters>,
 <300000, CarEnters>, <350000, CarEnters>,
 <450000, CarEnters>,  <550000, CarEnters>,
 <650000, TrainEnters>,  <900000, CarEnters>,
 <910000, CarEnters>, <920000, CarEnters>,
 <930000, CarEnters>,  <1000000, StopSim>)
```

Figure 5.1: The input data for the simulation of a normal situation

The simulator generates the following output when the first train (at $t = 40000$) is detected.

```
[SCH] (040000) : Received event 2002
[SCH] (040000) : Insert task 1000.
[EDF] (040000) >> TASK_START(tid = 1000,wcet = 240) <<

-----[ BGN READY QUEUE << 040000 >> ] -----
   nr | startt | rel.t | wcet | deadl | crt | taskid
   ---+--------+-------+------+-------+-----+-------
RT: 0 |  40000 | 40000 |  240 | 55000 |  Y  | 1000
      No entries in the ready queue....

    ======= BGN Time Table =======
**** No entries in the table ****
    ======= END Time Table =======

-----[ END READY QUEUE ] -----

[SCH] (040001) : Schedule task 1006 into ready queue.
```

Event 2002 is TrainEnters and task 1000 is *TrainIn* (see Table 4.1). Task 1006 (= *LowerGateTimeout*) is scheduled and stored in the Time Table, the tasks will be inserted in the Ready Queue after MaxLowerTime. Because this is the first train in the critical region, the execution time of the task is equal to its worst-case execution time, so the task finishes execution at time 40240.
Detection of a train and the execution of task *TrainIn* results in the following output of the environment (not the entire event list is given here).

```
[ENV] (040000) : Sending event < TrainEnters >
[TC ] (040000) : Start  execution of << TrainIn >>
[ENV] (040000) : Actuator signal 'LOWER THE GATES' received.
[TC ] (040240) : Finish execution of << TrainIn >>

----[ BEGIN ENVIRONMENT STATE (040240) ]----
Number of trains :    1
Number of cars   :    0
Gate status      : MVDOWN
Semaphore status : GREEN

Event List :  (<050000, TrainEnters>, <059229, GateClosed>,
               <145232, TrainLeaves>, <260000, CarEnters>,
               . . .
               <930000, CarEnters>, <1000000, StopSim>)
----[ END ENVIRONMENT STATE ]----
```

There is one train in the critical region, the gate is moving down and at $t = 059229$ the gate will be closed (see event GateClosed in the event list).
The load of the Ready Queue is calculated with Formula 5.1. The load of the Ready Queue measured during this simulation run is drawn in Figure 5.2. We can see that the load never reaches a value above 0.07. (Recall that only if the load is ≥ 1 an overload situation has occurred.
When we measure the width of the bars, we get the utilisation of the task processor, since a bar indicates that at least one task is running. The total time the processor was running is 6266 ms and the total simulation time was 1,000,000 ms. Thus the total processor utilisation is approximately 0.627%.
During the simulation at most one task at the time was ready for execution. This implies that, in this case, the response time of every task is equal to its execution time.

5.2.2 Exceptional situations

Overload situations

To examine overload situation we do not need to simulate. Because it can be calculate when an overload situation will occur with the method given in Section 3.4. The more general form of the equation can be used to calculate the load fraction of the partitions of the critical task set, namely

$$u = X \cdot u_{CarIn} + Y \cdot u_{TrainIn} \leq 1$$

to determine when an overload situation occurs. The two load fractions $u_{TrainIn}$ and u_{CarIn} were defined in Section 3.4 and calculated in Section 5.1.2.
If we consider only trains, i.e. $X = 0$, we see that $Y \leq 30$, i.e. when 31 trains are

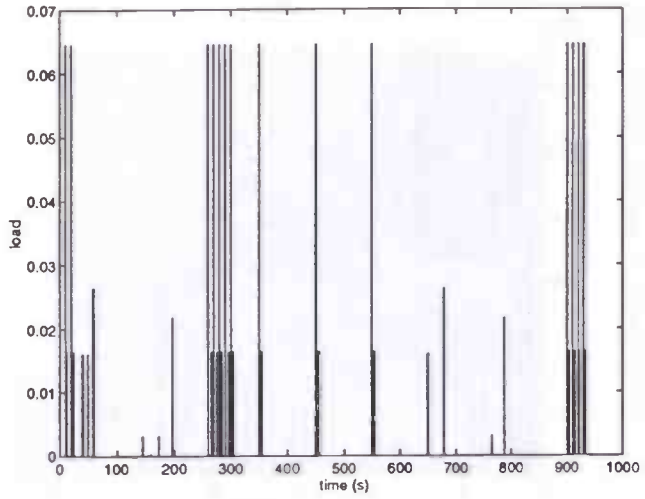


Figure 5.2: The load of the Ready Queue during the simulation of a normal situation

detected at the same time, this will cause an overload.
When only cars enter the crossing, i.e. $Y = 0$, we see that $X \leq 3$, we can handle three cars that arrive at the same time.
We know that at most two cars can arrive at the same time, therefore we take $X = 2$ and we see that we can handle maximal four trains at the same time.

High utilisation

Because we know that the load fraction of *CarIn* is high, compared to the load fraction of *TrainIn*, we simulated a situation where a car entered the crossing at high speed every 400 ms, i.e. the cars leave the crossing in about 2.2 seconds. After 33 seconds the simulation terminated. Figure 5.3 shows the load of the Ready Queue during this simulation. We see that the load, again, does not reach 0.07, but we also see that almost the entire duration of the simulation at least one task was present in the Ready Queue. During the simulation, the processor was used for 23049 ms, which means that the CPU utilisation is $23049/33000 \approx 69.8\%$.

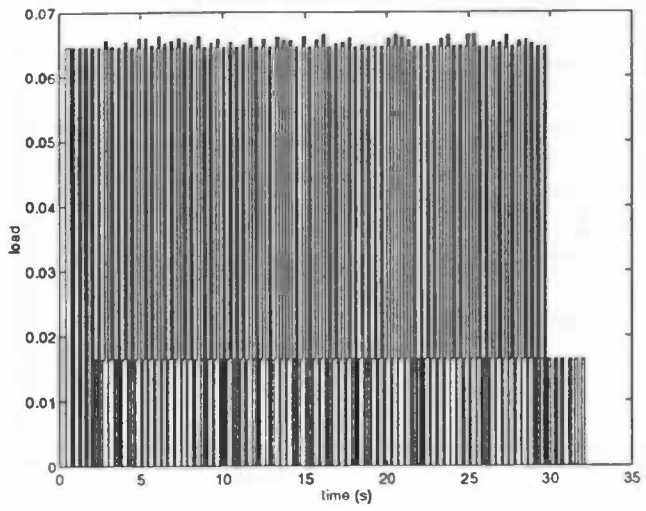


Figure 5.3: The load of the Ready Queue when every 400 ms a car is detected

Other exceptional situations

Finally, we simulated three other exceptional situations. First, a car entered the crossing, while the gates were closed. The environment gives the following output

```
[ENV] (027000) : Sending event < CarEnters >
[ENV] (027000) : Warning:
                 Car has crashed through the gate!
[TC ] (027000) : Start execution of << CarIn >>
[ENV] (027004) : Actuator signal 'SIGNPOST RED' received.
```

When the trains are stopped and the car has left the crossing, the trains can continue their way. Because one of the gates is broken, the crossing can not be controlled by the computer system until the gate is repaired.
In this situation we can not always avoid that trains collide with the car. If the car crashes through the gate just in front of the train, the train is not able to stop in time. This situation does not differ from the normal situation until the car crashes through the gate. For ARTIE this does not change anything. The load of the Ready Queue is still low, the CPU utilisation is also low and the response time of the tasks are equal to their execution times (see Figure 5.4).

We also simulated the situation in which a car took too long to leave the crossing. The gates were already closed and after seven seconds the task *CarSafetyMeasure* was executed. The environment generates the following output:

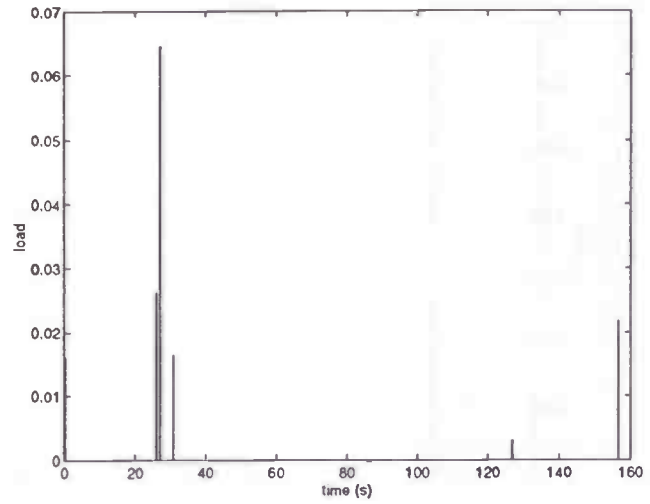


Figure 5.4: The load of the Ready Queue when a car has crashed through a closed gate

```
[TC ] (037006) : Start  execution of << CarSafetyMeasure >>
[ENV] (037006) : Actuator signal 'SIGNPOST RED' received.
[TC ] (037132) : Finish execution of << CarSafetyMeasure >>

----[ BEGIN ENVIRONMENT STATE (037132) ]----

Number of trains :    1
Number of cars   :    1
Gate status      : DOWN
Semaphore status : RED

Event List :  (<056732,  CarLeaves>, <200000, StopSim>)

----[ END ENVIRONMENT STATE ]----
```

In Figure 5.5 we can see that at time $t \approx 37$ seconds the load has a peak. Normally the load does not rise above 0.07, but now it almost reaches 0.35. This peak is caused by *CarSafetyMeasure*. This is a task with the deadline close to the worst-case execution time. This results according to Formula 5.1, in a higher load.

The final experiment is the case that the gates stay closed. We simulate the situation where, while the gates are moving up, a new train is detected. In this case, the environment generates the following output:

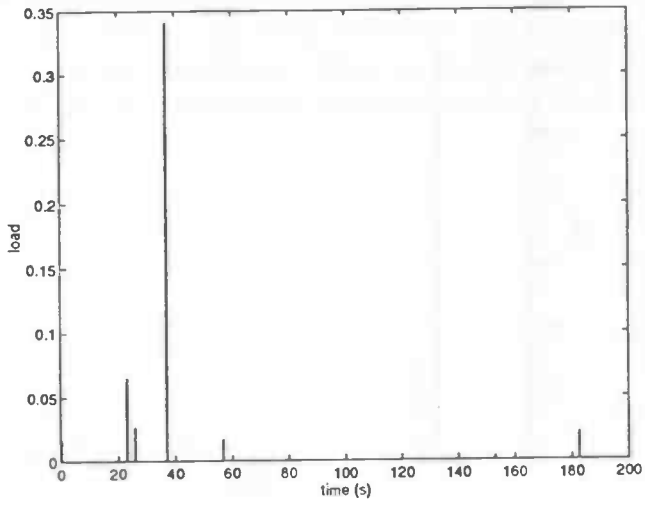


Figure 5.5: The load of the Ready Queue when a car got stuck on the crossing

```
----[ BEGIN ENVIRONMENT STATE (076480) ]----

Number of trains :    0
Number of cars   :    0
Gate status      : MVUP
Semaphore status : GREEN

Event List :  (<100000, TrainEnters>, <102342, GateRaised>,
               <200000, TrainEnters>, <300000, TrainEnters>,
               <550000, StopSim>)

----[ END ENVIRONMENT STATE ]----

[ENV] (100000) : Sending event < TrainEnters >
[TC ] (100000) : Start  execution of << TrainIn >>
[ENV] (100000) : Actuator signal 'LOWER THE GATES' received.
[TC ] (100240) : Finish execution of << TrainIn >>
```

In the setup of the simulations we assumed that trains are in the critical region at least 75 seconds and at most 150 seconds. When a train is detected every 75 seconds the gate will never be raised. In this situation the load of the Ready Queue is low, even lower than in the normal situation (see Figure 5.6). The task *CarIn* causes a load of ± 0.07 in the normal situation. But during this simulation cars do not have the chance to pass the crossing and so the load of the Ready Queue does not reach this value. The tasks executed during this simulation have more laxity between execution time and deadline.

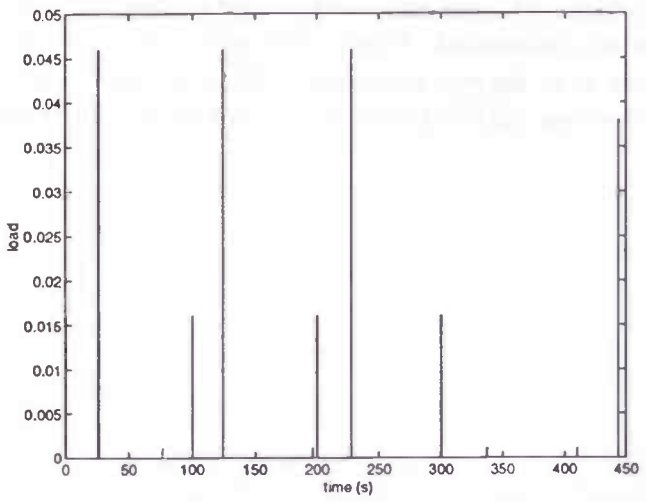


Figure 5.6: The load of the Ready Queue when cars never are able to use the crossing

5.3 Conclusions

We have run several simulation sessions to evaluate ARTIE. During these sessions, data concerning the behaviour of ARTIE were collected. The results of these experiments were presented and discussed.

The simulation of a normal situation was inspired by a real railroad crossing. Trains and cars arrived, and no accidents happened. No tasks miss their deadline and the CPU utilisation is low. During the simulation at most one task was ready for execution. Moreover, tasks started at the moment they were made ready for execution. So, their response time was equal to their execution time.

We have also presented the results of exceptional situations. We showed that overload can occur when 31 trains or four cars are detected at the same time. It is not possible to detect four cars at the same time, since cars only can arrive from two directions and there is always some distance between two cars at the same half of the road. Only two cars can be detected at the same time. In this situation we showed that four trains can be handled at the same time. Now we can conclude that this controller, with the given characteristics, is suitable for a crossing with four tracks.
It was assumed that cars do not drive faster than 80 km/h. In the situation that a number of cars enter the crossing with this speed we get a high CPU utilisation. The load of the Ready Queue still does not rise above the 0.07. It was seen that in this situation the processor was busy most of the time, but every task met its deadline. We

can conclude that even in this extreme situation ARTIE performed well.

During the simulation of three other exceptional situation it was seen that the behaviour of ARTIE was not influenced. Load, CPU utilisation and response time had almost the same values as in normal situations. Thus, we can conclude that in these three exceptional situations ARTIE performs as good as in normal situations.

Chapter 6

Conclusions

In this report we have discussed the development of a safety-critical application: a controller for a railroad crossing. The purpose of this application was to create a test case for ARTIE, an architecture for real-time systems. Simulation sessions were used to collect data about ARTIE's behaviour.

We used the CASE-tool EPOS for the specification and design of our application. In the specification language EPOS-R we created two documents: one to define the requirements for the railroad crossing and one to work out a solution concept. Next, we used EPOS-S to convert the solution concept into a system design.
The railroad crossing is a rather small application, there are not many events that can occur and there are not many tasks. Besides, the application was developed by not more than two persons. These are the reasons why we didn't use all parts of EPOS. EPOS is particularly suited to develop larger systems in which more persons are involved. Especially in these cases it is useful to extensively use the tool for requirement engineering and project management.
In our situation EPOS-S was very useful to design the application. But it was hard to express the difference between a task that is ready for execution and a task that is assigned to the task processor. When a task is called, it is interpreted as storing the task into the Ready Queue. To remove a task from the Ready Queue the EPOS-S instruction `STOP` (*task*) is used. Using DA-FE, the graphical front-end tool of EPOS, it is easy to create and maintain a system design. If changes are made in the specification of the application, they can be easily handled using DA-FE.
In the future it is maybe possible to setup a large project where several people can be working on. More parts of EPOS can be used than we have done. For example, EPOS-P, for project management, can be used. Also more requirements, constraints and terms can be defined. At the end of the development process EPOS-A can be used to analyse the design.

After the design of the application with EPOS, the tasks were formally specified and validated using wp-calculus. Because wp-calculus is not especially suited as a real-time specification language, Statecharts were used to formulate preconditions, postconditions

and an invariant of the system.
We used Statecharts to describe the behaviour of the railroad crossing. In this way it was easier to derive an invariant of the system. Using the Statecharts we could express the postconditions of the tasks in terms of states. With the postconditions and the code of the tasks we showed, using wp-calculus, that the system does not violate the invariant.
The responsibility of the application programmer is to prove that the set of critical tasks is feasible for ARTIE. None of the critical tasks may exceed its deadline. With the method that was given in [Tol95] we showed under which circumstances the set of critical tasks is feasible.

Using the simulation of the Sections 5.2 and 5.3 it can be concluded that ARTIE performed well with the railroad crossing as application. During the simulations we have seen that the Ready Queue often did not contain any tasks, thus the processor was often idle. We have also observed that the time between two events is in general long.

The execution times are relative large considering the code of the tasks. In other words, we have simulated a rather slow processor. Because ARTIE performed well with this slow processor, we can conclude that with a faster processor the results may be even better. Because the tasks will finish earlier execution, the CPU utilisation will be lower.
A suggestion for future research is to take another real-time application to test ARTIE. This new application should be even more time-critical than the railroad crossing controller. The new test case should have smaller deadlines, deadlines more close to the execution times of the tasks. We can think of cruise controllers in cars or controllers in planes (fly by wire).

Bibliography

[BBvKT94] Ton Biegstraten, Klaas Brink, Jan van Katwijk, and Hans Toetenel. A simple railroad controller: A case study in real-time specification. Report 94-86, Technische Universiteit Delft, 1994.

[BBWF95] C.M. Bailey, A. Burns, A.J. Wellings, and C.H. Forsyth. A performance analysis of a hard real-time system. 1995.

[BdG93] Kees Bosma and Erik de Groot. Evaluation of some scheduling algorithms for real-time systems. Master's thesis, Rijks*universiteit* Groningen, March 1993.

[Cha94] Roderick Chapman. Programming timing analysis. Dependable Computing Systems Centre University of York, May 1994.

[DF84] E.W. Dijkstra and W. Feijen. *Een methode van programmeren.* Academic Service, 1984.

[EPO89] Gesellschaft für Prozeßrechnerprogammierung mbH, Kolpingring 18a, 82041 Oberhaching bei München. *EPOS-Documentation*, 1989.

[Gom93] Hassan Gomaa. *Software Design Methods for Concurrent and Real-Time Systems.* SEI series in software engineering. Addison-Wesley Publishing Company, Inc., 1993.

[GYF94] Vered Gafni, Amiram Yehudai, and Yishai A. Feldman. Activation-oriented specification of real-time systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems.* Springer-Verlag, September 1994.

[Hen89] R. Henn. Feasible processor allocation in a hard-real-time environment. *Journal of Real-Time Systems*, 1:77–93, 1989.

[HJL93] C. Heitmeyer, R. Jeffords, and B. Labaw. Benchmark for comparing different approaches to specifying and verifying real-time systems. In *IEEE workshop on Real-Time Operating Systems and software*, 1993.

[HS91] W.A. Halang and A.D. Stoyenko. *Constructing Predictable Real Time Systems.* Kluwer Academic Publishers, Boston-Dordrecht-London, 1991.

[HT94] W.H. Hesselink and R.M. Tol. Formal feasibility conditions for earliest deadline first scheduling. Submitted. Also available as CSReport 9406, Dept. of CS *University* of Groningen, The Netherlands, 1994.

[JM86] F. Jahanian and A.K-L. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 9(12):890–904, September 1986.

[Lau89] R. Lauber. *Prozeßautomatisierung.* Springer-Verlag, second edition, 1989.

[Lau90] R. Lauber, editor. *EPOS Overview, Short Account on the Main Features.* Gesellschaft für Prozeßrechnerprogrammierung mbH, Kolpingring 18a, D-82039 Oberhaching bei München, December 1990.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 1(20):46–61, 1973.

[Tol95] Ronald M. Tol. *Formal Design of a Real-Time Operating System Kernel.* PhD thesis, Rijks*universiteit* Groningen, 1995.

[TtW94] R.M. Tol and G. te Winkel. Some scheduling experiments with ARTIE. In *Conference Proceedings of the IFAC/IFIP Workshop on Real-Time Programming.* Pergamon Press, June 1994.

Appendix A

Complete system design in EPOS-S

A.1 The action module-object

```
ACTION MODULE Controller.
DECOMPOSITION:
  Initialisation;
  WHILE TRUE
     DO Detect OD.
ACTIONEND.
```

A.2 Action-objects

```
ACTION Initialisation.
DECOMPOSITION:
CODE: "
  NumCars = 0;
  NumTrains = 0;
  GateStatus = "UP";
  SemaStatus = "Green"".
ACTIONEND.
```

```
ACTION Detect.
DECOMPOSITION:
  PARALLEL (
    WAIT FOR TrainEnters
      THEN TrainIn WAITEND,
    WAIT FOR TrainLeaves
      THEN TrainOut WAITEND,
    WAIT FOR GateClosed
      THEN GateIsClosed WAITEND,
    WAIT FOR GateRaised
      THEN GateIsUp WAITEND,
    WAIT FOR CarEnters
      THEN CarIn WAITEND,
    WAIT FOR CarLeaves
      THEN CarOut WAITEND ).
ACTIONEND.
```

```
ACTION TASK TrainIn.
DECOMPOSITION:
  IncTrains;
  IF NumTrainsIsOne THEN
    SET (GateClosed);
    Lower;
    WAIT FOR GateClosed
    WITHIN MaxLowerTime THEN
      GateIsDown;
      IF NOT NumCarsIsZero THEN
        CarSafetyMeasure;
      FI
    ELSE
      LowerGateTimeout;
    WAITEND
  FI.
TRIGGERED: TrainEnters.
ACTIONEND.
```

```
ACTION TASK TrainOut.
DECOMPOSITION:
  DecTrains;
  IF NumTrainsIsZero THEN
    SET (GateRaised)
    Raise;
    WAIT FOR GateIsOpened
    WITHIN MaxRaiseTime THEN
      "Skip"
    ELSE
      RaiseGateTimeOut;
    WAITEND
  ELSE "Skip"
  FI.
TRIGGERED: TrainLeaves.
ACTIONEND.
```

```
ACTION TASK CarIn.
DECOMPOSITION:
  IncCars;
  IF GateStatIsDown THEN
    IF SemaStatIsGreen THEN
      StopTrain;
    FI
  FI.
TRIGGERED: CarOnCrossing.
ACTIONEND.
```

```
ACTION TASK CarOut.
DECOMPOSITION:
  DecCars;
IF NumCarsIsZero THEN
  IF SemaStatIsRed THEN
    RestartTrain;
  ELSE STOP (CarsafetyMeasure);
  FI
FI.
TRIGGERED: CarFromCrossing.
ACTIONEND.
```

```
ACTION TASK GateIsDown.
DECOMPOSITION:
  STOP (LowerGateTimeOut);
  RESET (GateClosed);
  IF NumCarsIsZero THEN
    STOP (CarSafetyMeasure);
  FI
  IF SemaStatIsRed THEN
    RestartTrain;
  FI.
TRIGGERED: GateClosed.
CODE: "GateStat = "DOWN"";
ACTIONEND.
```

```
ACTION TASK GateIsUp.
DECOMPOSITION:
  STOP (RaiseGateTimeout);
  RESET (GateRaised).
CODE: "GateStat = "UP"".
TRIGGERED: GateRaised.
ACTIONEND.
```

```
ACTION TASK CarSafetyMeasure.
DECOMPOSITION:
  IF SemaStatIsGreen THEN
    StopTrain;
  FI.
ACTIONEND.
```

```
ACTION TASK LowerGateTimeout.
DECOMPOSITION:
  IF SemaStatIsGreen THEN
    StopTrain;
  FI.
ACTIONEND.
```

```
ACTION TASK RaiseGateTimeout.
DECOMPOSITION:
  RESET (GateRaised).
OUTPUT: "The gates have not raised!".
ACTIONEND.
```

```
ACTION PROCEDURE StopTrain.
CODE: "SemaStat = "Red"".
OUTPUT: Red TO
        SemaphoreInterface.
ACTIONEND.
```

```
ACTION PROCEDURE RestartTrain.
CODE: "SemaStat = "Green"".
OUTPUT: Green TO SemaphoreInterface.
ACTIONEND.
```

```
ACTION Lower.
CODE: "GateStat = "MVDOWN"".
OUTPUT: CloseGate TO
        GateInterface.
ACTIONEND.
```

```
ACTION Raise.
CODE: "GateStat = "MVUP"".
OUTPUT: OpenGate TO GateInterface.
ACTIONEND.
```

```
ACTION IncTrains.
CODE: "NumTrains = NumTrains + 1".
ACTIONEND.
```

```
ACTION DecTrains.
CODE: "NumTrains = NumTrains - 1".
ACTIONEND.
```

```
ACTION IncCars.
CODE: "NumCars = NumCars + 1".
ACTIONEND.
```

```
ACTION DecCars.
CODE: "NumCars = NumCars - 1 ".
ACTIONEND.
```

A.3 Event-objects

```
EVENT TrainEnters.
INTERRUPT.
TRIGGERS: TrainIn.
EVENTEND.
```

```
EVENT TrainLeaves.
INTERRUPT.
TRIGGERS: TrainOut.
EVENTEND.
```

```
EVENT CarEnters.
INTERRUPT.
TRIGGERS: CarIn.
EVENTEND.
```

```
EVENT CarLeaves.
INTERRUPT.
TRIGGERS: CarOut.
EVENTEND.
```

```
EVENT GateClosed.
INTERRUPT.
TRIGGERS: GateIsDown.
EVENTEND.
```

```
EVENT GateRaised.
INTERRUPT.
TRIGGERS: GateIsUp.
EVENTEND.
```

A.4 Data-objects

```
DATA NumTrains.
TYPE: FIXED.
RANGE: 0 -> 2.
DATAEND.
```

```
DATA NumCars.
TYPE: FIXED.
RANGE: 0 -> 6.
DATAEND.
```

```
DATA GateStatus.
TYPE: CHAR.
DATAEND.
```

```
DATA SemaStatus.
TYPE: CHAR.
DATAEND.
```

A.5 Condition-objects

```
CONDITION NumCarsIsZero.
CODE: "NumCars == 0".
CONDITIONEND.
```

```
CONDITION NumTrainsIsOne.
CODE: "NumTrains == 1".
CONDITIONEND.
```

```
CONDITION SemaStateIsRed.
CODE: "SemaStatus == "Red"".
CONDITIONEND.
```

```
CONDITION SemaStatIsGreen.
CODE: "SemaStatus == "Green"".
CONDITIONEND.
```

A.6 Interface-objects

```
INTERFACE GateInterface.
INTERFACEEND.
```

```
INTERFACE SemaphoreInterface.
INTERFACEEND.
```

Appendix B

The code of the tasks

```
TASK TrainIn
  numtrains = numtrains + 1;
  IF numtrains == 1 THEN
    ENABLE(gate_closed);
    SIGNAL(gate, lower);
    IF gatestatus == mvup THEN
      REMOVE(RaiseGateTimeout);
    FI;
    gatestatus = mvdown;
    INSERT(LowerGateTimeout)
      AFTER max_lower_time;
    IF numcars != 0 THEN
      INSERT (CarSafetyMeasure);
    FI;
END; /* TrainIn */
```

```
TASK TrainOut
  numtrains = numtrains - 1;
  IF numtrains == 0 THEN
    ENABLE(gate_raised)
    SIGNAL(gate, raise);
    gatestatus = raise;
    INSERT(RaiseGateTimeout)
       AFTER max_raise_time;
  FI;
END; /* TrainOut */
```

```
TASK CarIn
  numcars = numcars + 1;
  IF gatestatus == down  THEN
/* Car crashes through the gate */
    IF semastatus == green THEN
      SIGNAL(sema, red);
      semastatus = red;
    FI;
  FI;
END; /* CarIn */
```

```
TASK CarOut
  numcar = numcar - 1;
  IF numcars == 0 THEN
    IF semastatus == red THEN
      SIGNAL(sema, green);
      semastatus = green;
    ELSE REMOVE(CarSafetyMeasure);
    FI;
  FI;
END; /* CarOut */
```

```
TASK GateIsUp
  REMOVE(RaiseGateTimeout);
  DISABLE(gate_raised);
  gatestatus = up;
END; /* GateIsUp */
```

```
TASK RaiseGateTimeOut
  DISABLE(gate_raised);
  MESSAGE(The gate
          has not raised!);
END; /* RaiseGateTimeout */
```

```
TASK GateIsDown
  REMOVE(LowerGateTimeout);
  DISABLE(gate_closed);
  gatestatus = down;
  IF numcars == 0 THEN
    REMOVE(CarSafetyMeasure);
  FI;
  IF semastatus == red THEN
    SIGNAL(sema, green);
    semastatus = green;
  FI;
END; /* GateIsDown */
```

```
TASK CarSafetyMeasure
  IF semastatus == green THEN
    SIGNAL(sema, red);
    semastatus = red;
  FI;
END; /* CarSafetyMeasure */
```

```
TASK LowerGateTimeout
  IF semastatus == green THEN
    SIGNAL(sema, red);
    semastatus = red;
  FI;
END; /* LowerGateTimeout */
```

Appendix C

Manual for the simulator

C.1 Running a simulation

C.1.1 Directory structure

The files for simulator are stored into two directories: scheduler and controller. All the files that are used by the kernel can be found in the directory scheduler. The implementation of the task processor and the application can be found in the directory controller.
The input-file `sim_env_descr.dta`, see Chapter 4, is read by the process simulating the environment of the application. It should be stored the directory controller.
To compile the simulator, execute the command

```
make -f Makefile
```

in each directory. In the directory scheduler the executable file `scheduler` is generated. In the directory controller the executable file `controller` is made.

C.1.2 Input

To run a simulation session, the simulator has to know when events occur. The user has to give the simulator the input by creating a file, `sim_env_descr.dta`, which contains event identifiers and the points in time at which the events occur. The user has only to define the occurrences of the events TrainEnters and CarEnters. The simulator of the environment generates the other events like GateRaised and TrainLeaves.
Every line of the file contains a pair of numbers. The first number is the time, in milliseconds. The second is the event identifier, for example 2 = TrainEnters and 4 = CarEnters (see Table 4.1 on page 34). Figures C.1 shows us the contents of the input file for the normal situation as presented in Section 5.2.1.

```
 10000  4
 20000  4
 40000  2
 50000  2
260000  4
270000  4
280000  4
290000  4
300000  4
350000  4
450000  4
550000  4
650000  2
900000  4
910000  4
920000  4
930000  4
1000000 e
```

Figure C.1: Example of the input-file `sim_env_descr.dta`

In the last line of the file the user expresses when the simulation run has to be terminated. The event identifier 'e' is an extra event, it does not belong to the application.

A user can now easily create new simulation sessions by replacing the current events and points in time by others.

C.1.3 Simulating

The simulator must be started by creating two processes. The first process is created by executing the scheduler program. When this process is running the controller program has to be executed. The simulation has now begun. When the simulation is finished, both processes are terminated.

C.1.4 Output

During the simulation, two different types of output-files are generated. In the files `sched.rep` and `contr.rep`, stored in scheduler and controller respectively, the output of the simulator can be found. The output of the scheduler shows when tasks are scheduled and executed, when events occurs and the contents of the Ready Queue, see Figure 4.2 on page 32. The output of the controller shows when tasks are executed,

when events are generated and the status of the environment, see Figure 4.3 on page 34. The data gathered during the simulation are stored in the files `stat.load`, `stat.resp` and `stat.util` in the directory scheduler. `Stat.load` shows the data about the load of the Ready Queue during the simulation. When a task is inserted into or removed from the Ready Queue, first the old value of the load and then the new value is stored in the file. The first column (`time`) shows the time when a change took place, the second (`load`) shows the load values, see Figure C.2.

```
----[ BEGIN LOAD ]----

  time   |  load
---------+----------
 000000  | 0.000000
 000007  | 0.000000
 000007  | 0.064545
 000234  | 0.064545
 000234  | 0.019373
 001345  | 0.019373
 001345  | 0.327624
 012265  | 0.327624
 012265  | 0.000000

----[ END LOAD ]----
```

Figure C.2: Example of the output-file `stat.load`

Figure C.3 depicts an example of the output-file stat.resp. In this file the response times of the tasks that were executed during the simulation are stored. The taskid, the response time and the starttime of the tasks can be found.

```
----[ BEGIN RESPONSE TIME ]----

 tid   | resp. time | exec. time
-------+------------+-----------
 100a  |   000007   |   000007
 1004  |   000265   |   000288
 100b  |   000335   |   000385
 1004  |   000988   |   000988

----[ END RESPONSE TIME ]----
```

Figure C.3: Example of the output-file `stat.resp`

In tt stat.util the first column (`CPU use (ms)`) shows how long the task processor was used (see Figure C.4). The second column (`Total time`) shows how long the simulation has runned. The CPU-utilisation of a finished simulation can be determined by dividing the value of the first column in the last row by the value of the second column on the last row.

```
----[ BEGIN UTILISATION ]----

 CPU use (ms) | Total time
--------------+-----------
    000143    |  00010143
    000308    |  00012430
    000451    |  00020143
    000616    |  00025153

----[ END UTILISATION ]----
```

Figure C.4: Example of the output-file `stat.util`

C.2 Replacing an application

Recall from Chapter 4 that the simulator consists of the following parts:

- An implementation of ARTIE. The following two parts are implemented:
  - The kernel processor, running the kernel programs
  - The task processor, running the application
- A simulator of the railroad crossing's environment

We can see that two parts of the simulator contain 'application dependent' program code. The task processor executes the tasks of the application and the simulated environment is a model of the railroad crossing.

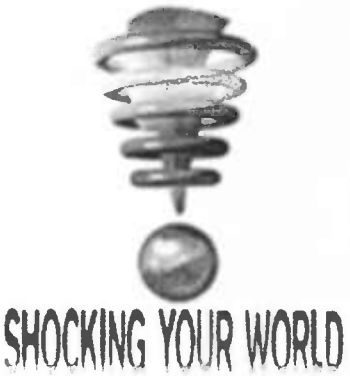
In the file `task_code.c` (in the directory controller) the tasks and the functions for calculating the execution times of the tasks are implemented. In this file also the Task Table and the Event Table are initialised.
If a user of the simulator wants to replace the application of the railroad crossing by another, he has to put the code of the tasks in this file, make new Event- and Task Tables and create new functions to calculate the execution times of the new tasks.

In the file `environment.c`, in the directory controller, the environment of the railroad crossing is simulated. For example, if an approaching train is detected, the environment generates the TrainLeaves event. This part of the program also updates the status of the environment when a change has taken place.
With a new application new events are to be generated and a different state has to be updated.

MACROMEDIA — about the company

1996 MACROMEDIA EUROPEAN USER CONFERENCE

DECEMBER 4 - 6, 1996 * RAI CENTRE * AMSTERDAM * THE NETHERLANDS



Register by Email

Don't miss out! Register Now!
Email: m.media@nsc.nl

General Management

Tony Gautier
Reed Exhibitions Netherlands
PO Box 470
1000 AL Amsterdam
The Netherlands
Tel: +31 20 515 9282
Fax: +31 20 515 9279
Email: reedex@pi.net

Conference

Martin de Weerd and Judith Mooner
Nederlands Study Centre
PO Box 330
3130 AH Vlaardingen
The Netherlands
Tel: +31 10 434 99 66
Fax: +31 10 434 32 67
Email: m.d.weerd@nsc.nl or j.moonen@nsc.nl