Vrije Universiteit Brussel
Programming Technology Lab
Faculty of Sciences
Department of Computer Science
Academic Year 2000 - 2001

A Finite State Machine Approach to Real-Time Scheduler Generation for Embedded Systems

Jessie Dedecker
jessie.dedecker@vub.ac.be

Promotor: Prof. Dr. Theo D'Hondt

Thesis submitted in partial fulfillment of the requirements for the degree of Licenciaat in de Informatica

Abstract

Component-based development is a paradigm that explicitly emphasizes the reusability problem. Components are software pieces that are plugged together to form a program. Each component can be seen as a reusable part that can be plugged in other software compositions. Unfortunately embedded software has special characteristics (such as robustness, timing behaviour, ...), which compromise the reusability of such software. Therefore, components are not adapted to the characteristics common to embedded software. A real-time system runs software in which the correctness of the program depends not only on the logical results, but also on the time at which the results are produced. Timing constraints are used to specify the temporal corectness of the software. A real-time system is often concurrent by nature. A scheduler determines which concurrent process has priority over the others to keep up with the timing constraints. We use an adapted component model in which all component communication is asynchronous. This thesis proposes a way to extract semantic data from the component source code. The extracted data can be converted to a deterministic finite automata (DFA) which can be used by a finite state machine (FSM) to track the progress of the software. The information of the tracker can be used to make scheduling decisions on a single-processor embedded system. The techniques presented in this dissertation are a first step towards the reusability of software components in real-time embedded software.

Contents

List of Figures

List of Tables

Acknowledgements

This dissertation would not be what it is today, without the enormous amount of support that was given to me. Therefore I wish to express my gratitude towards:

Prof. Dr. Theo D'Hondt for promoting this dissertation

Werner Van Belle who came up with the subject and who helped me through every stage of this work from preparation through implementation and writing to proofreading. His comments and professional advise were very supportive and constructive.

Dr. Kim Mens, Dr. Tom Mens and Johan Brichau for proofreading my work and giving me valuable advise on both the content and the linguistics of this dissertation.

Holger Kenn for giving me advise about the time analysis of the component system.

The researchers of the Programming Technology Lab for listening and giving comments during the weekly presentations.

My parents for giving me the opportunity to study in the best possible circumstances and for being my best supporters.

The Vrije Universiteit Brussel for the excellent education.

Part I

Describing Real-Time Behaviour

Chapter 1

Embedded Systems

Embedded systems exhibit certain characteristics such as timing behaviour and high reliability demands, which have a great influence on the software development cycle. This chapter discusses these characteristics and how they have an impact on the development of embedded software. The chapter concludes with the exact contribution of this dissertation and an outline of the other chapters in this dissertation.

1.1 Definitions

The word "embedded" can be used in different contexts which are loosely linked together, but each have a distinct meaning. As is common in the area of computer science there are often different versions of definitions for each term. The definitions we will use here are taken from [1, 2].

Definition 1 (Embedded Processor) *Special features in respect to power-consumption, i/o-ports, and / or price of a processor are often used to denote a processor as being "embedded".*

E.g., most of the microprocessors that are based on an ARM core are embedded processors.

Definition 2 (Embedded System) *A computational device, which is strongly subject to physical constraints in its functional behaviour. It can for example be linked to a mechanical system or a chemical process, requiring timely responses obeying natural laws. Not only time, but also other factors like for example energy can be severe physical constraints.*

E.g., an autonomous mobile robot.

Definition 3 (Embedded Software) *Embedded software is software that runs on one or more processors embedded in a product and that is inextricably bound up with this product and its functionality. It cannot be bought separately, and it constitutes a large added value for the product.*

E.g., software used in a mobile phone.

1.2 Characteristics

Embedded systems can be recognized because they exhibit some characteristics that distinguish them from most desktop applications.

Robustness and Reliability Embedded systems have high reliability and robustness demands: Depending on the task performed by an embedded system, a failure in this system can be the cause of serious damage. E.g. the software that controls the airbags in your car, the software in a missile guiding system. Also, embedded software is difficult if not impossible to upgrade. Furthermore, the lifetime of embedded systems is often very long, which adds to the demands of robustness.

Time Constraints An embedded system typically interacts continuously with the surrounding environment and has to do this in a time-constrained manner. Most embedded systems exhibit a real-time behaviour. The very nature of real-time embedded applications makes certain characteristics of their implementation (such as timing and implementation architecture) critical. Usually, the software in these applications is responsible for the control of other equipment (e.g., mobile robots need real-time software to control the motor-sensor interactions).

Most real-time embedded systems are, by nature, multitasking solutions to real-world problems. The different parts of these systems usually run at different priorities and with different run-time characteristics. The notion of multiple tasks or threads being active in the system at the same time is common. Some of these real-time systems are deployed on a set of microprocessors in a distributed architecture.

Because real-time systems are the main topic of this dissertation we provided some clear definitions of the terms that are going to be used in the remainder of this dissertation:

Definition 4 (Real-Time System) *a Real-time System runs software in which the corectness of the program depends not only on the logical results, but also on the time at which the results are produced.*

E.g. a mobile robot.

Definition 5 (Hard Real-Time System) *a Hard Real-Time System is a real-time system in which the lack of adherence to timing constraints results in a system that becomes useless and could result in a catastrophic system failure.*

E.g. propulsion system of a space shuttle.

Definition 6 (Soft Real-Time System) *a Soft Real-Time System is a real-time system in which the ability to meet deadlines is indeed required, but failure to do so does not render the system useless, nor does it cause a system failure.*

E.g. a network router.

Resource Constraints Another important characteristic of embedded systems is the wide variety of critical metrics that must be taken into account in the software. These metrics include throughput, latency, program and data memory requirements, energy consumption, and financial cost.

1.3 Evolution in the Way of Developing

A lot of progress has been made to the area of developing desktop application over the last decades, while the evolution of software development for embedded systems has made little evolution, because embedded software was so small and not interesting from a research point of view. Embedded software becomes complexer, because advanced processors and more memory becomes available at cheaper prices.

1.3.1 Current Practices in Embedded Software Development

A lot of embedded software developers are still writing their software in low level languages such as C and assembly code, which makes the produced code error-prone, unreliable, difficult to read and therefore difficult to maintain. In a lot of embedded software development there is little or no reuse of existing code.
This is caused by several reasons of which some are mentioned here:

Abstraction comes with a cost Many developers believe that the use of data abstraction comes with a cost in performance. In the early days this assumption was true, but over the years the compilers have evolved and have become more efficient with each new release. Processors also provide more computational power at the same costs compared to a few years back.

Diversity of hardware used in embedded devices Code developed for a particular embedded system sometimes cannot be reused because a different chip architecture is used and the code does not compile for the new platform.

1.3.2 Opportunities for Code Reuse

The opportunities for software reuse depend on the available resources in the embedded system. We can distinguish three different kinds of embedded systems based on the available resources.

- *Severely constrained embedded systems* have extremely limited resources available and perform mostly obvious tasks. Because of the limited memory it becomes impossible to use any high-level language and most of the software is developed using assembly code and C code. The functionality of such an embedded device is often also implemented in hardware instead of software. Example: a simple digital watch.

- *Moderately constrained embedded systems* have typically more complex software, which adds to the needs of having more resources available. The need for more complex software is mostly caused by the need of extra features on certain embedded devices. Examples: a television set, a set-top box, a digital camera, ...

- *Loosely constrained embedded systems* have resources available that can be compared or even exceed that of a regular Desktop PC. The development cycle of software for such an embedded system cannot be compared to that of a Desktop PC, because they are still inextrically bound with the hardware of the device. Besides that, the characteristics of robustness, high reliability demands and timing-behaviour still exist. Examples: control systems for nuclear power plants, control systems for industrial manufactoring.

It is obvious that there is little opportunity for code reuse in *severely constrained embedded systems* because these programs are so small due to the severe limitations on memory usage and CPU power. On the other hand *moderately constrained* and *loosely constrained embedded systems* could benefit from using a software development methodology that has reuse foremost in mind.

1.4 Problem Definition

In the early days embedded systems were so small, that reuse of existing software was practically impossible. So with each new embedded system a new version of the software was written. Advanced processors and memory became cheap over the years, the embedded systems became more complex as well as the software. This increased complexity required research on a higher level for embedded systems.

Edward A. Lee [8] reduced most of this research to one question:

> "How do we adapt the software abstractions designed merely to transform data to meet requirements like real-time constraints, concurrency and robustness?"

The SEESCOA (Software Engineering for Embedded Systems using a Component-Oriented Approach) consortium tries to find an answer to the question stated above using an adapted component-oriented approach. Members of this consortium are research teams from different flemish universities. There are two research groups of the VUB that participate in this consortium, namely the Programming Technology Lab and the System and Software Engineering Lab. Chapters 2 and 3 describe this adapted component-oriented approach and contain information from the deliverables [3] of the SEESCOA project.

In this dissertation we will focus on how we can do *soft* real-time scheduling for this adapted component-oriented approach, while trying not to harm the reusability principles of the independent components. Scheduling decisions will be made using semantic data extracted from the source code.

1.5 Approach

This section describes how this dissertation is organised. Figure 1.4 gives a total view on the outline of this dissertation.

- Chapter 2 gives a short introduction to components.
- Chapter 3 describes the implementation of the component system that is used throughout this dissertation and how it is adheres to the adapted component definition [10] proposed by the SEESCOA consortium. This chapter reflects the global view shown in figure 1.4.
- Chapters 4 and 5 propose a formal documentation format to describe the communication behaviour of the components. This is shown in figure 1.4 as the component documentation that is given as input to the tracker and the scheduler.
- In chapter 6 we discuss how to extend the documentation with information that is common to the characteristics of embedded software.
- Chapter 7 discusses measuring the execution times of components. It also describes how the documentation can be used together with the execution times to analyze the temporal behaviour of the software.

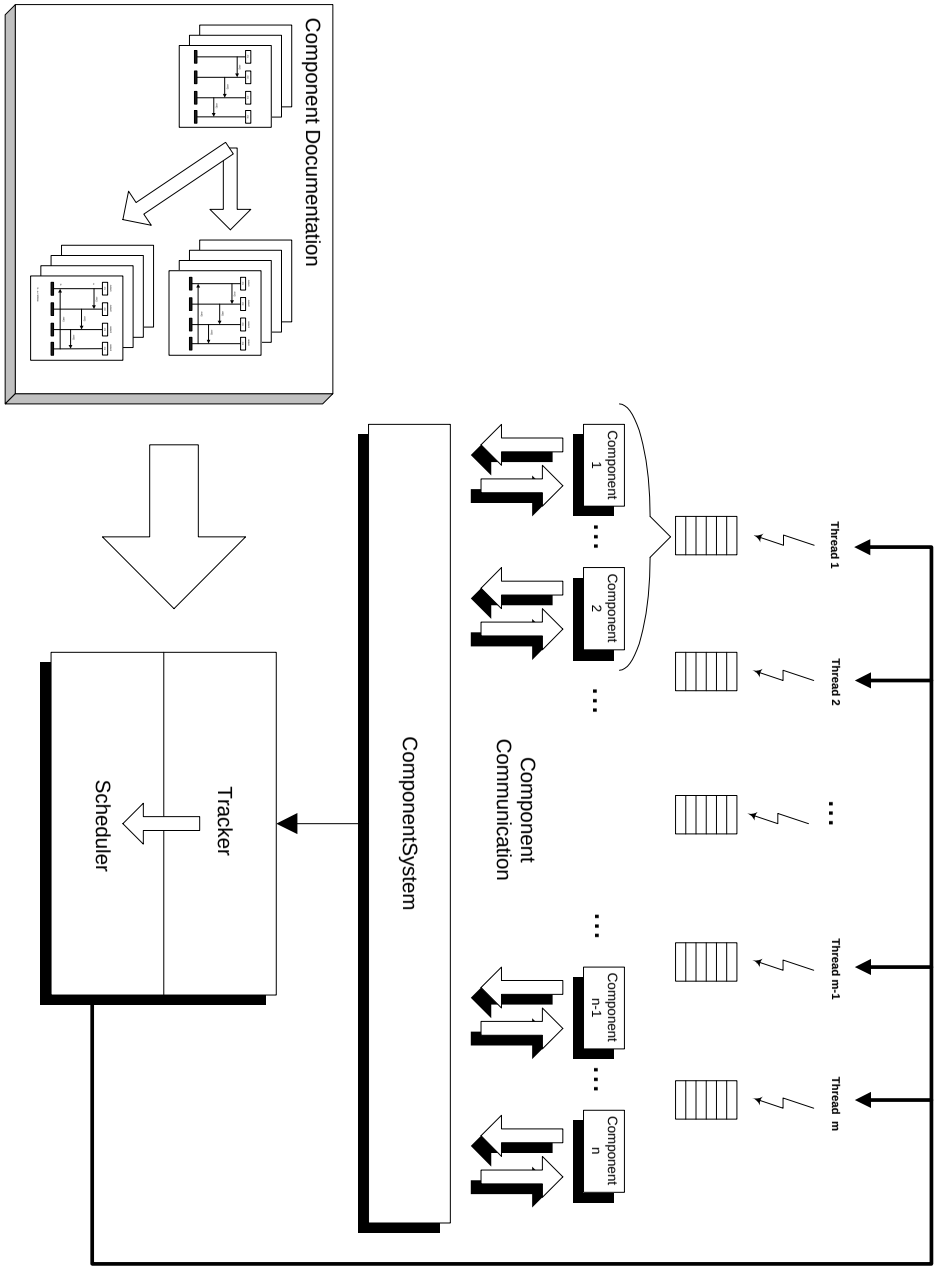

Figure 1.1: Outline of this dissertation

- In chapter 8 we give an algorithm to follow the execution of a running system by matching the component interaction with the documentation. The tracker is encorporated in the scheduler component in figure 1.4.
- Chapter 9 describes how the tracker is used to enforce the temporal behaviour within the component system. In figure 1.4 this is represented with the line between the scheduler and the different threads.
- In chapter 11 we discuss how to reduce the overhead by reducing the number of threads in the component system, without affecting the temporal constraints of the embedded system. Represented with the brace between the threads and components in figure 1.4.
- Chapter 12 is a reiteration of the previous chapters. This chapter also discusses future work.

1.6 Summary

Existing paradigms that support code reuse do not take the characteristics that are specific to embedded systems into account. The opportunities for code reuse depend on the available resources in the embedded system. This dissertation focusses on how we can make soft real-time scheduling decisions by using semantic data extracted from the source code.

Chapter 2

Components and Reusability

The latest paradigm that is used for software reuse is called *component based development* (CBD). Remember from chapter 1 that most research about embedded systems is to find an adapted software abstraction so that it meets the characteristics of the embedded systems. In this dissertation we adapt the CBD paradigm to meet these characteristics. Component based development is given preference to object-oriented development because a component is explicitly designed for reuse, which is not always the case for objects. Components are analogous to hardware components. For example, when a new television set is put together and the manufacturer chooses to replace the analogue tuner with a digital one, then the idea is to replace the software component controlling the analogue tuner with another version supporting a digital tuner. The adaptation of the software to the new hardware design can then happen by replacing one component, while reusing the other software components.

2.1 Definitions

Throughout the rest of this dissertation we use some terms related to components. Because there exist many definitions and terms about components we describe what is meant by them in this section.

Definition 7 (Component) *A component is a reusable documented software entity that is used as a building block for software systems. It is used to perform a particular function in a specific application environment within a specific component system. Components are composed (glued together) using their interfaces. These interfaces consist of provided interfaces and required interfaces.*

As stated in the definition above a component has two kinds of interfaces: provided interfaces and required interfaces:

Definition 8 (Provided Interface) *A Provided Interface describes the code signatures that must be used to access the services provided by the component.*

Some components cannot provide their services without accessing services of other components.

Definition 9 (Required Interface) *A required interface describes what services a component needs to be able to provide its services.*

E.g. a component providing webservices needs the services of a network component to send the requested data over the network.

In the component definition a distinction has to be made between an abstract component and a concrete component.

Definition 10 (Abstract Component) *An abstract component is a description of a reusable software element; it does not have a state. It also does not make sense to talk about the runtime properties of an abstract component.*

Definition 11 (Concrete Component) *A concrete component is an instantiation of an abstract component and has run-time properties.*

The term component is more general; by using it we mean both aspects.

Definition 12 (Interface Provider) *An interface provider is a component that is capable of providing the services defined by an interface, also sometimes called the implementor of an interface.*

Definition 13 (Client) *A component A that is using the services provided by component B is called a client of component B.*

2.2 Components vs. Objects

Concrete components are not objects. And in consequence, abstract components are not classes. First of all, components cannot inherit from other components. Objects do inherit from each other (they inherit the implementation). Abstract components are also extensively documented, which is not always the case for classes. Note that in some cases classes are documented with diagrams, semantics, call protocols, and so on, but this information is not explicitly described in the object definition. This information is often added in an informal way. When a component is implemented, it will probably use different objects to perform its functionality (of course in the case an OO language is chosen). Therefore some books talk about components as if they were big objects. This is true to some extent, but limiting the

component definition to this would be wrong. A concrete component should be thought of as having its own code and data space and also its own thread of control. This is necessary to have the ability to use different synchronization principles and make components reusable. Thinking a component has its own control flow will be more general than a component which enforces certain calling strategies upon other components. Or, a component written in the assumption its memory will be accessed by other components is more specific than a component which does not share its data using these kinds of techniques.

A component (the abstract as well as the concrete component) is always used in a certain application environment and in addition it also offers an application environment to its users.

2.3 Plugging Components Together

When creating a program we need to puzzle over the different interfaces of the components. In a fully standardized world this could be done by just matching the correct interfaces of the different components. The real world, however, is not standardized at all, which implies that the component manufacturers invent names for the interfaces as they are needed. Because of this, different interfaces are provided for accessing the same services. Another problem that we are faced with is that when two components are communicating, data is passed along in the form of parameters. The format or types of these parameters needs to match as well. To overcome the problems stated above we need a bit of code to convert the parameters when components interact with each other. To match the different interfaces we can use the adapter pattern as described in [16]. The extra code that is needed to have a flawless communication between the components is called *glue code*.

2.4 Summary

Software components are reusable software entities. They have the explicit guarantee that they are reusable in other software compositions. Components are however not adapted to the characteristics common to embedded software. In this chapter we have described how the components are adapted to match these characteristics.

Chapter 3

The Component System

This chapter starts by giving a rough definition of a component system. Afterwards we discuss what services should be provided by a component system for use in embedded systems. We conclude this chapter by looking more closely at how some of these services are satisfied in the component system that is used throughout this dissertation. Because component communication is one of the main services a component system provides we discuss on the different communication methods that can be used in a component system. As most embedded systems are concurrent, we talk about how concurrency can be introduced into the component system. Finally we discuss what services the component system provides in order to create a real-time system.

3.1 Definition

The component system is the infrastructure (framework, architecture or kind of operating system), which makes component instances work together, which glues them and creates a homogenous environment for them. The component system can be seen as the middleware that connects different components and which makes them work together. To use a metaphor: the component system provides the streets while the components are the cars driving on it.

3.2 Tasks of a Component System

This section gives a brief overview of the services a component system could provide to the components running on the system. These services are not always a requirement, it depends on the nature of the embedded system onto which the component system has to run. This is why we make the distinction between the required *base services* and the *optional services*.

Base Services

The component system

- Makes components work together. The component system should create and destroy component instances and be able to start and stop component instances.

- Abstracts the hardware and the operating system such that all components run in the same environment. For example, if we work in an embedded system with segmented memory or in a system with five flavors of memory access, the component system should solve this and offer a more or less flat interface to it. This abstraction should be as lightweight and as performant as possible in embedded systems. The component system should be mapped upon the operating system and programming language as closely as possible. It is not said that all hardware dependent issues should or could be put into the component system. All general hardware aspects that have impact on all of the code (like the memory access example) should be put into the component system. Modular hardware access, such as devices, can be put into separate components.

- Handles message passing between components: If a component wants to make another component do something, or whenever the state of another component has to be changed, a message is send to the component in question. Components must be able to send messages to other components using a reference (which can be obtained by using the unique name of the component). The component system takes care of sending data (over a network for example), calling the right function on components and eventually other ways of passing messages between components. This includes changing the data format if necessary, as is done in CORBA. Nevertheless, the component system is not necessarily a distributed environment.

- Provides some standard glue components to adapt interfaces between different components. For example, a certain component can return a callback with a specific name, whilst the receiver expects the message with another name. This can be handled by certain glue components.

- Intercepts hardware and software interrupts and models them as sending a message to the appropriate component handling the interrupt.

Optional Services

- Can have support for introspection. When working with components we need the ability to find, name and rename components. These

abilities could be provided by the component system. Furthermore, sometimes it is necessary that a client can query a component about its services. Mostly the client is bound to the components interface at client construction time (e.g. when the client is compiled). When introspection is possible, the client is not bound at client construction time, but it can dynamically (at runtime) find the services of a component. This can be compared to the reflection mechanism in Java and Smalltalk.

- Handles the scheduling between components. Because components are thought of as active entities it is necessary to map this view to a real operating environment. This is done by the component system, which ensures priorities of messages between components, which takes care of (hard) real-time constraints and scheduling in general.

3.3 Component Communication

Components need to interact with each other for different reasons such as requesting services, notification of an event, etc... This happens through component communication, which is a base service of the component system. In this section we give a brief overview of how the different components can communicate with each other.

3.3.1 Shared Memory vs. Message Passing

There are basically two ways in which components can communicate with each other: through shared memory and by sending messages.

Shared Memory

In a shared memory architecture the different components have a common memory region in which data can be written and read (see figure 3.1). When component A wants to communicate information to component B it writes data to the shared memory and this data is then read by component B. Component B does not know when component A will have some data for him, so it will poll the memory at certain times. When memory is accessed by different concurrent processes, different situations can occur that can make the data in memory inconsistent. For example, when component A is writing data to the shared memory and component B starts writing data in the same memory region, the data could become inconsistent. Problems as these are solved by placing a lock before accessing the shared resource and removing the lock when the component is done accessing the shared resource. Placing locks is a good way of avoiding such problems, but makes the code prone to situations such as deadlocks and livelocks. Another disadvantage is

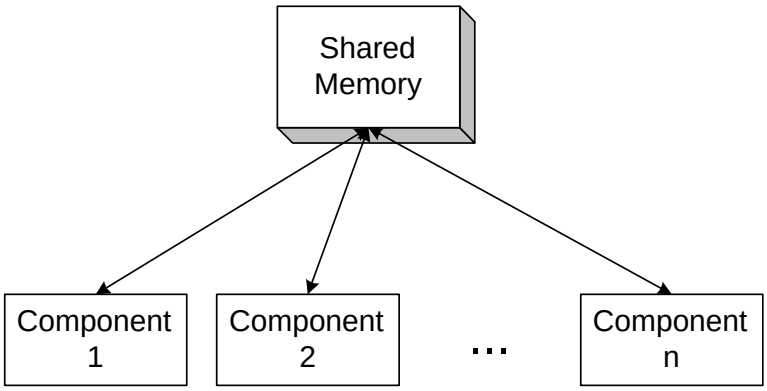

Figure 3.1: Shared Memory

	Shared Memory	Message Passing
Efficiency	+	-
Distributed access	-	+
Reusability	-	+

Table 3.1: Comparison of Shared Memory vs. Message Passing

that shared memory is hard to use in a distributed environment as memory is bound to a machine. There is some research done to use memory in a distributed environment [4]. One could also argue that the usage of shared memory conflicts with the idea of reusability. When we have a look at the different paradigms that were invented we see that shared memory is tried to be kept to a minimum. In procedural languages the use of local variables was recommended as much as possible. In object-oriented languages a step further is made by putting the data space with the operations onto that data into one encapsulated entity.

Message Passing

When component A needs to communicate with component B it could also send a message to component B. This message would then contain the information for component B. Message passing has the same overhead as a regular procedure call. When a message is passed, the parameters are put onto the stack and some call is made to component B.

3.3.2 Synchronous vs. Asynchronous messages

Considering the different advantages and disadvantages of shared memory and message passing, the choice was made to do all component commu-

nication through message passing. All component communication is done through the communication layer of the component system. This layer is necessary to lookup the components' location on the system or on another system in a distributed environment. The layer is also used to provide information to the scheduler. There are two ways in which components could send messages among each other:

- *Synchronous* messages can be compared with regular method calls, so when component A sends a message to component B at some point, then component A is interrupted, component B processes the message and afterwards component A resumes (Figure 3.2 (a) on page 23). This kind of behaviour is the typical behaviour of most object-oriented and procedural programming languages.

- *Asynchronous* messages happen if component A sends a message to component B and component A does not wait until B has processed this message before continuing. This implies that component A and B can process their messages concurrently. Figure 3.2 (b) on page 23 shows an asynchronous method call on a single processor system. Asynchronous messages are used in actor-based languages [5].

The component system used in this thesis uses an asynchronous message model for all the intercomponent communication. Such a message model introduces concurrency into the component system implicitly. This has a number of advantages over an explicit thread based system:

Communication between concurrent parts happens through explicit messages and not through a shared data space. As noted earlier a shared resource can introduce subtle errors in a program.

Semantic information can easily be extracted from messages. We will explain later (see chapter 4) why this is important and how this can be done.

Mapping Using messages allows a customised mapping of components onto threads. Chapter 11 is dedicated to this subject.

Safeness Asynchronous messages are safer when used in distributed environments. When a computer crashes in synchronous communication, then the system is waiting for data that never comes resulting in a locked system.

But an asynchronous message model also has some drawbacks that need to be considered:

Return values When an asynchronous message m is sent by component A to component B, then component A does not wait until component B has processed that message and cannot get a return value. One of the reasons component A could have send a message to component B is to compute some value and then return that value. Hence when component B has computed that value it can send a message back to component A containing the return value. The problem now is that, while component A was awaiting that return value, it might have processed other messages that altered its state, making it impossible to use the return value in further computations. Situations like these are resolved by adding a special kind of parameter to the message m, namely a hidden parameter [6]. The semantics of a hidden parameter are as follows:

- When a component processes a message with a hidden parameter, then this parameter is send along with all messages that are sent while processing that message.
- Only the component that has put the hidden parameter in a message can access that parameter. This means the parameter is hidden for all other components.

Now when a component A sends message m to component B, it can put the state of that computation in a hidden parameter. When component B sends a return message with the computed value, the hidden parameter will be sent along. Component A can now access the hidden parameter, restore its state and continue with that computation.

Extra code Each time a component sends a message to a component that sends a message back with a return value, that component needs to provide code that:

1. Saves the state of the computation
2. Accepts the message containing the return value
3. Restores the state the computation

It is obvious that the extra amount of code causes the programming cycle to be prolonged and creates computational overhead.

Order of the messages When different asynchronous messages are sent to a component, then there is no guarantee in what sequence they will arrive at that component. This can cause several problems such as race conditions. Consider following example: A component provides access to a certain socket on a network interface. The component provides an interface to do the following three operations: OpenSocket, WriteToSocket and CloseSocket. A

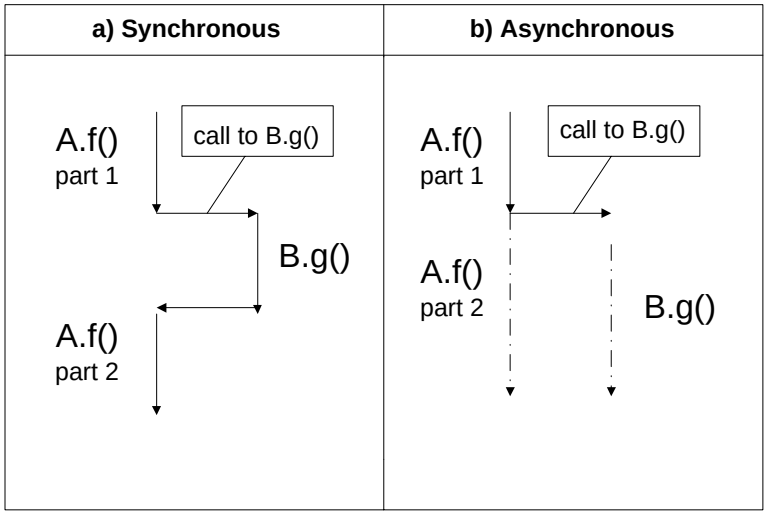

Figure 3.2: Synchronous (a) vs. Asynchronous (b) messages

component accessing this component will typically first send an OpenSocket message followed by several WriteSocket messages and eventually request to close the socket. This example shows that the order of the incoming messages cannot be altered and should be preserved at all times. The component system assures that the order of the incoming messages is preserved.

3.4 Levels of Concurrency

In embedded systems we often want to create a system that is able to react to events that happen in its environment. Most of the time the embedded system is performing other tasks when these events occur. When a system has to react to these events within a time bound we cannot always wait until the system has finished the other tasks. This is why many embedded systems have concurrent software. The Octopus methodology [7] describes three ways for introducing concurrency into an object-oriented system in a single processor architecture. Figure 3.3 comes from [7], but is adapted to components. These three distinctions can also be made in a component-based system:

1. *Component interaction concurrency*

2. *Intracomponent concurrency*

3. *Intercomponent concurrency*

3.4.1 Component Interaction Concurrency

When a message for a component is processed, the component system chooses the message that is going to be processed next. After processing that message, another message is chosen for processing. Two messages cannot be processed simultaneously, so this is actually a single-threaded system. It is important to note that, when a message is being processed, it must complete before another message can be processed. Such behaviour is called non-preemptive (see figure 3.3). *Component Interaction concurrency* cannot be used in Real-Time Systems because it lacks pre-emptive concurrency. Pre-emptive concurrency is a requirement for a Real-Time System. It is obvious that when the processing time of a message is relatively long and a new time-constrained message occurs, the system is unable to process the message within the time boundaries. (Also see figure 3.3).

3.4.2 Intracomponent Concurrency

In intracomponent concurrency a single component can process multiple messages concurrently (see figure 3.3). this kind of flexibility is not recommended because it can introduce subtle errors into the system (such as deadlocks, livelocks, etc...). This kind of concurrency also works by default on shared resources, which is the source of many of these problems mentioned above. These problems make the extendibility of a system difficult without introducing concurrency errors into the system.

3.4.3 Intercomponent Concurrency

When a component is processing a message and a *different* component receives a message, then the message that is currently being processed can be interrupted or *pre-empted* to process the message received by the other component first (see figure 3.3). The distinction between *component interaction concurrency* and *intercomponent concurrency* comes from the fact that a message can be pre-empted for another message in intercomponent concurrency. *Intercomponent concurrency* is the kind of concurrency that is used in the component system to avoid the problems that were stated above. In a system using intercomponent concurrency the use of shared resources is avoided where possible. Because each component has its own resources it becomes easier to distribute the components onto different embedded systems that can communicate with each other.

3.5 Internal Structure of the Component System

In section 3.3.2 we have advocated our choice for asynchronous communication between the different components. This section discusses how asyn-

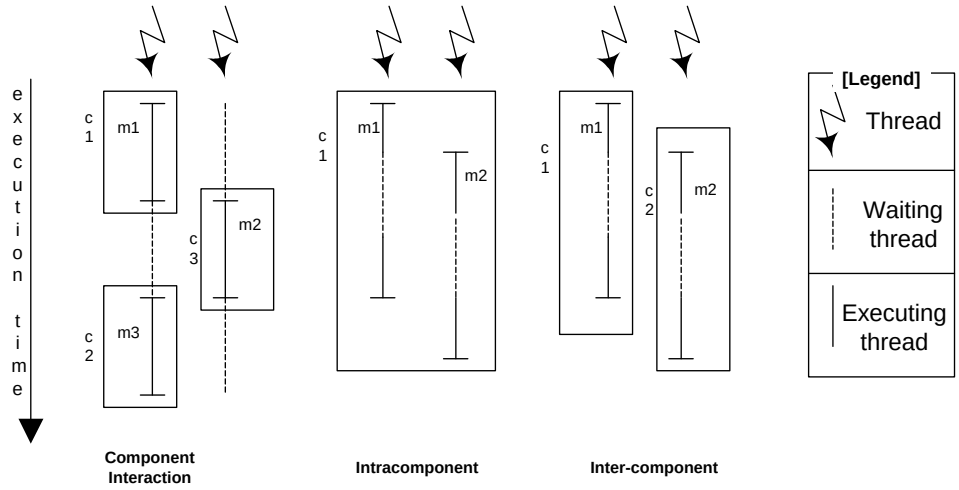

Figure 3.3: Levels of Concurrency

chronous communication is achieved in the component system. Each component has a queue in which its incoming messages are placed. When component A sends a message m to component B, then the message m is placed in the queue of component B.

There is a difference between the *logical model* and the *implementation model*. The logical model represents how the user of the component system should think about the component system, while the implementation model refers to the actual implementation of the component system. In the logical model each component has its own thread and consumes the messages that are in its queue. The scenario described above is shown in figure 3.5. In chapter 11 however, we see that it is possible to group several components onto a single thread. This is referred to as the implementation model.

We have developed a precompiler that translates the asynchronous message calls into calls to the underlying component system. An example of component-code is shown in figure 3.4. The example shows a component that acts as a logging device. The message *Log()* retrieves the object stored in the hidden parameter called *Output* and calls the method write upon it. The parameter *Data* is passed to the method write. Finally the message *Logged()* is send to the caller of the message *Log()* to inform that the data is logged. Note that an synchronous message is send with a ".." instead of a "." that is used for method calls. Besides the difference for asynchronous messages that are marked with a ".." there is also special syntax for retrieving and passing parameters with asynchronous messages. This syntax is summarised in table 3.2.

```
componentclass Logger
{
  <<Other code>>

  message Log()
  {
    >BufferedWriter|Output<.write(<String|Data>);
    <Return>..Logged();
  }

  <<Other code>>
}
```

Figure 3.4: Example message in Component Code

Syntax	Semantics
<x>	retrieve the message parameter x
<y|x>	retrieve the message parameter x and typecast it to y
<x:y>	put the message parameter x with value y
>x<	retrieve hidden parameter x
>y|x<	retrieve hidden parameter x and typecast it to y
>x:y<	put hidden parameter x with value y

Table 3.2: Parameters Passed with Asynchronous Messages

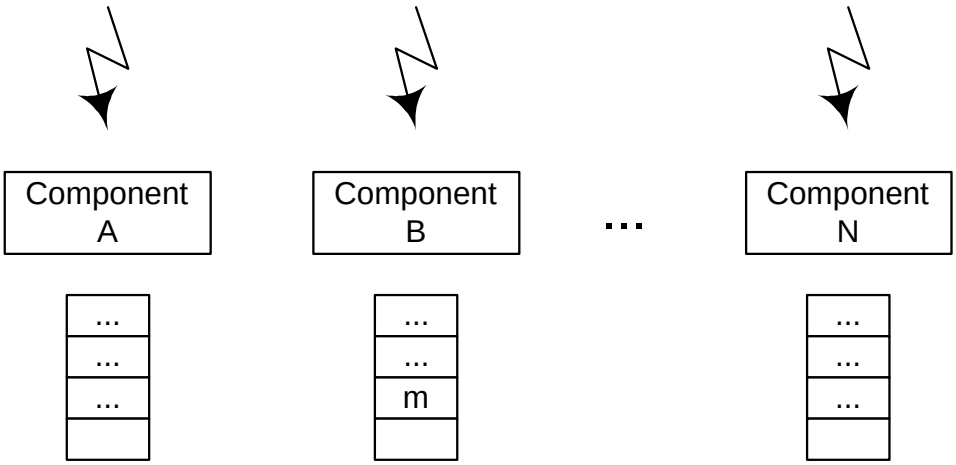

Figure 3.5: Logical Model of the Component Communication

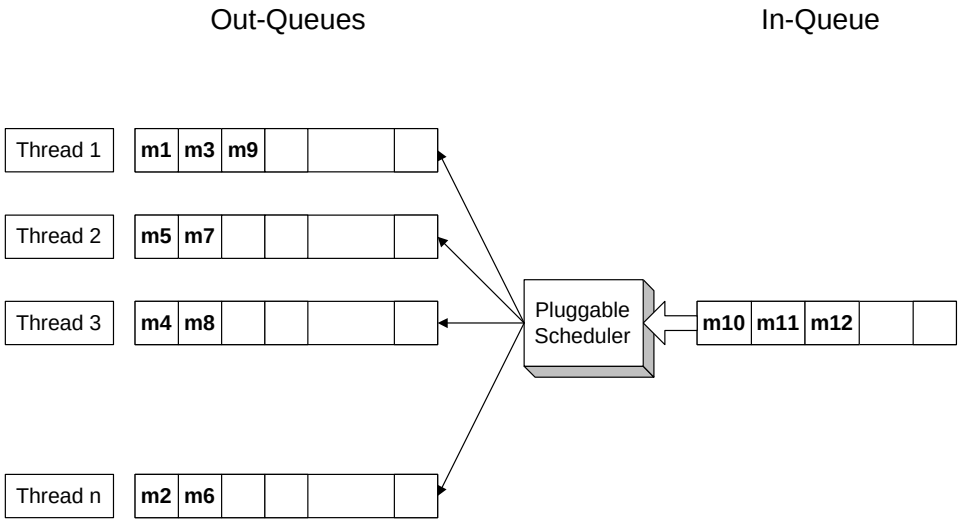

Figure 3.6: Pluggable Scheduler

3.6 Scheduling in the Component System

As described in section 3.4 embedded systems need to handle events coming from the environment. Because an embedded system cannot react to events when it is busy doing other things, we needed concurrency. One thing that was left out from that story is that some of these events need to be handled within some timeframe. Consider the embedded software that needs to open your airbag when you have a car crash. When the system only opens your airbags a few seconds after the car crashes, then they are of no use.

To handle events within a certain time frame, the component system needs to process more important messages before other less important messages. The process of choosing which processes have priority over others in order to handle an event within a certain timeframe is done by a *real-time scheduler*. Chapter 9 discusses some techniques on how to choose these processes and introduces a novel technique which chooses these processes by using semantic data extracted from the source. The component system allows different schedulers to be used (see Figure 3.6), because each embedded system has its own characteristics. When a message is sent by a component it is placed in a queue to be handled at some point in time. The scheduler has to decide which queue is chosen to enqueue the message. The scheduler can also manipulate the different priorities of the threads according to the deadlines that needs to be met. In chapter 11 we discuss how we can reduce the number of threads in a program.

3.7 Summary

The component system provides services to the components. One of the most important services the component system provides is the component communication. The component communication is used to introduce inter-component concurrency into the system in an implicit manner. Some events in the component system must be handled within a certain time frame. A scheduler tries to enforce these time constraints by changing the priorities the different threads. We can also schedule the system by chosing one of the queues when a message is send. In the remainder of this dissertation we present a scheduler that can be plugged into the component system to enforce the temporal behaviour of the software.

Chapter 4

Describing the Abstract Behaviour of a Program

Description languages are used to describe a description a piece of code in a formal way. These description languages can be used to analyze specific properties of the code. Description languages are used in this thesis, because they can help in making scheduling decisions (see chapter 8 and chapter 9). Message sequence charts were initially [11] created to describe specific communication scenarios for use in telecommunication hardware and software. In this chapter we discuss how they can be used and adapted to describe communication scenarios in a program using components. But first, a short introduction to message sequence charts is given.

4.1 Message Sequence Charts

4.1.1 Introduction

A Message Sequence Chart (MSC) does not describe the complete behaviour of a system, it merely expresses a specific execution trace of a program. If we want to have a complete execution trace of a system we have to combine different MSCs together to express the full behaviour of a system. MSCs are in process of standardisation [11] and have many extensions that are out of the scope of this thesis. We discuss *basic MSCs* and how these basic MSCs can be combined to glue the different scenarios together using *High-level MSCs*.

4.1.2 Basic Message Sequence Chart

Describing Communication Behaviour

A Basic Message Sequence Chart (bMSC) expresses a *partial* description of the communication behaviour between different *instances*. An instance

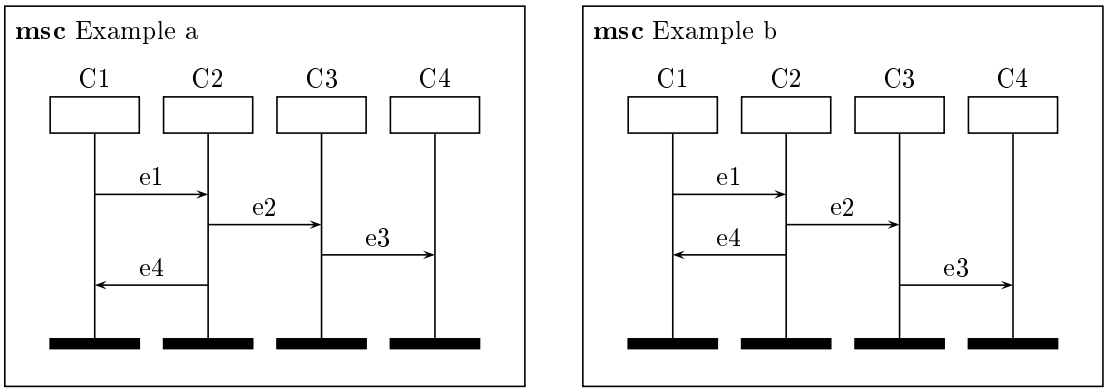

Figure 4.1: Example of two equivalent basic MSCs

is an abstract entity of which one can observe the interaction with other instances. Instances can communicate with each other through message-sends. The messages are send asynchronously. An instance is denoted on figure 4.1 as a vertical axis and messages are denoted by an arrow between two vertical axices. Figure 4.1 shows 4 instances interacting with each other. C1 sends a message e1 to C2. Upon arrival of e1, C2 sends a message e2 to C3. When e2 arrives at C3 it sends a message e3 to C4. The order of the messages can be determined by assuming that each instance has its own time starting from the top to the bottom. The time between the different instances can then be determined by applying the transitive properties of the time. E.g., e1 arrives before e2 and e3 is received after e2. We cannot always determine a total order of time between the different instances. It is for example uncertain when e4 is send. e4 could be send before e3 is send or after e3 is send. Example a and b of figure 4.1 are equivalent to each other.

Describing Alternative Compositions

Another interesting feature, described in [11], is that you can describe alternative compositions within the basic MSCs. This is done through *inline operator expressions*. Graphically, the inline operator is described by a rectangle. The operator expression is placed in the upper-left corner of the rectangle. There are 5 inline operator expressions available:

par Parallel execution of certain parts in the basic MSC. A dashed line is used to mark what parts are executed in parallel.

loop Describes an iteration.

opt Optional execution means that this region could be executed once or not at all.

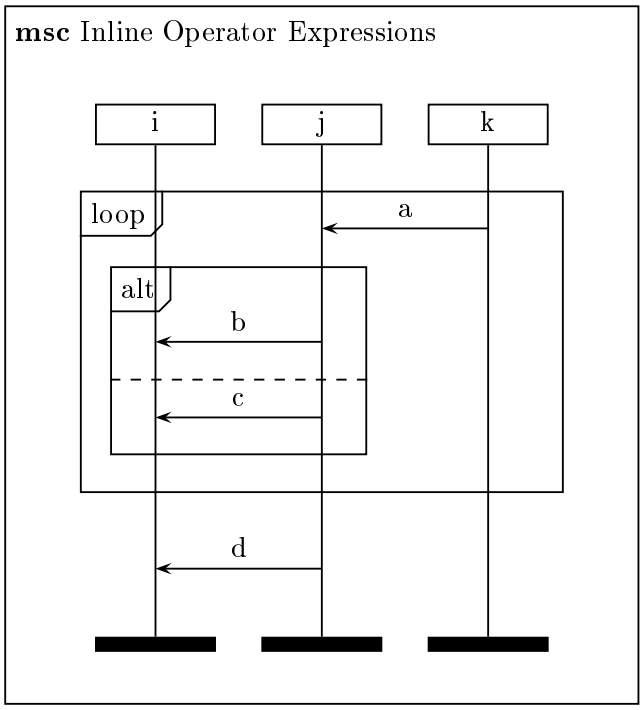

Figure 4.2: Example of a basic MSC using inline operator expressions

alt Alternative executions are marked with the alt. The possible execution regions are seperated with dashed lines. When entering such a section each region describes an alternative basic message sequence chart.

exc Exceptions.

As is shown in figure 4.2 we also can nest different inline operator expressions to describe more complicated scenarios. The MSC describes that in the loop instance k sends the message a to instance j. After instance k sent that message instance j sends either message b or message c to instance i. When the loop terminates the message d is send from instance j to instance i.

4.1.3 High-Level Message Sequence Chart

A basic message sequence chart only describes one possible execution in a part of the program. However, we want to describe all possible executions of a certain program. To be able to describe a full program we must combine several basic MSCs together. This is exactly what is possible with *High-level Message Sequence Charts* (hMSCs). The available constructs are shown in figure 4.3.

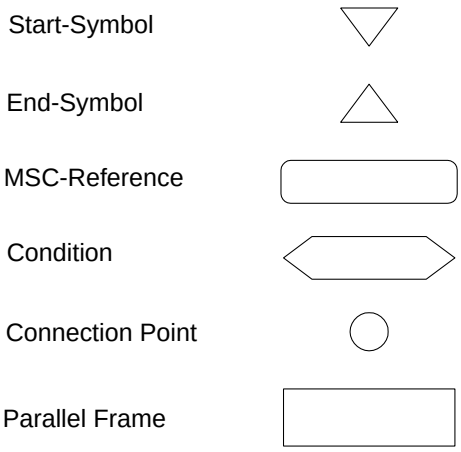

Figure 4.3: Building Constructs of High-Level MSCs

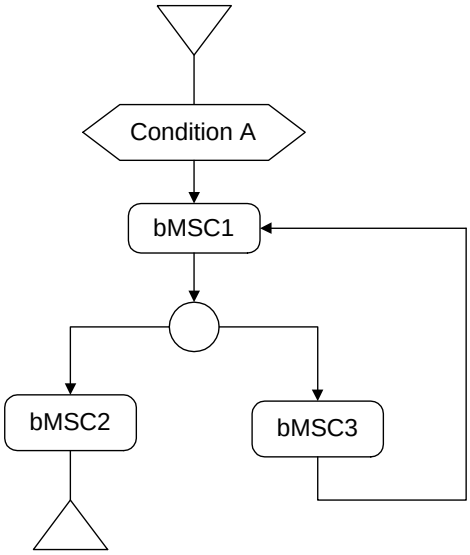

Figure 4.4: Example of a High-Level MSC

The example shown in figure 4.4, describes that when *Condition A* is satisfied that:

1. the scenario described by bMSC1 occurs and we proceed either with step 2 or 3.
2. scenario bMSC2 occurs and the system terminates.
3. bMSC3 occurs and we go back to step 1.

4.2 Describing the Behaviour of a Program using MSCs

A message is handled by executing some internal code and by requesting services from other components by using the required interfaces as described in chapter 2. To do scheduling we are particularly interested in the inter-component communication, because that is where there is an introduction of concurrency into the system as explained in chapter 3. In [15] MSCs are also used to formally verify if different components can interact with each other. Throughout the remainder of this dissertation we will use MSCs to describe the *communication* behaviour of components. So, when we talk about the behaviour of a component we actually refer to the communication between components. An instance can be viewed as a component instance (denoted by a vertical axis in figure 4.2) and the communication (denoted with an arrow between two component instances in figure 4.2) are the messages that are sent among the components. Note that this is a valid mapping because the component system described in chapter 3 sends its messages asynchronously.

4.2.1 Problems with MSCs

As MSCs were not initially intended to describe the full communication behaviour of a program we encountered several problems describing components:

1. Parameters are not expressed in MSCs. In component communication we can have two different parameters: regular parameters and hidden parameters.
2. MSC documentation does not always reflect the actual code. MSCs are mostly used to specify scenarios that need to be resolved by the software. The actual implementation of the software does not always reflect the MSCs that were specified.
3. Because glue code can be very dynamic it can become very hard to describe some pieces of source code.

4. A MSC merely describes one execution trace, while we want all possible execution traces.

5. A full trace of the communication behaviour can only be given after that the components are glued together and the abstract components become concrete components, this is necessary for tracking the software.

6. Instances that automatically send messages cannot be expressed. Such information is interesting for the scheduler, because that is where we have an introduction of computation into the system.

In the remainder of this chapter we will discuss how we have extended the specification of MSCs to overcome the problems mentioned above.

To solve problem 2 we have made changes to the component precompiler that was mentioned in section 3.5 to output adapted MSCs that reflect the exact source code of the component. This has the advantage that the adapted MSCs will always reflect the latest version of the component.

4.2.2 Abstract MSCs vs. Concrete MSCs

When we consider the source code of a component we are considering the *abstract behaviour* of a program. We can only consider the full or *concrete behaviour* when the components are composed together in a program. Also note that the concrete behaviour describes the run-time aspects of the program, which is the behaviour we want to capture. In the remainder of this chapter we will discuss how we can describe the *abstract behaviour* from the component's source code. The discussion how we can extract the *concrete behaviour* from the *abstract behaviour* is placed in chapter 5. It is important to note that we make the distinction between MSCs that describe *the abstract behaviour*, which will be called *abstract MSCs* and MSCs that describe *the concrete behaviour*, which are called *concrete MSCs*. The MSCs shown in the remainder of this chapter are all abstract MSCs.

4.3 Describing the Abstract Behaviour of a Component

To describe the abstract behaviour of one component we will discuss how each message of a component is described.

4.3.1 Dealing with Programming Language Constructs

When describing the behaviour of source code we have to describe the regular constructs that are provided in the programming language. In a programming language we have control flow constructs such as conditions (if-

Control Flow Structure	Inline Operator Expression
if-then	opt
if-then-else	alt
switch-case	alt
for-loop	loop(x,y)
while-do	loop(0,Max)
do-while	loop(1,Max)

Table 4.1: Mapping of Control Flow Structures to Inline Operator Expressions

then-else statements, switch statements, ...) and loop structures (for-do, do-while, while-do, ...). All these control flow structures determine if and when messages are send and how many messages are send. All these possibilities should be described by the MSCs. To describe these control flow structures we are going to use a subset of the *inline operator expressions* (loop, opt, alt) that were introduced earlier in this chapter. Table 4.1 shows how these constructs can be described using inline operator expressions. Please note that we only describe the constructs in which component communication occurs. If we have an *if-then-else* statement with component communication in the *else-brach*, but not in the *then-branch*, then we must use an *opt* construct instead of an *alt* construct. Examples can be found in appendix A.

4.3.2 Describing Abstract Exceptions

In some programming languages another construct exists that can influence the control flow of a certain program, namely *exceptions*. They were left out of the discussion in the previous paragraph, because they are described somewhat differently. A typical use of exceptions is demonstrated in figure 4.5. There is no straight-forward approach to handle them using previously discussed inline operator expressions. In the next chapter we see that the exceptions are expanded to an *alt* construct. Figure 4.6 shows an abstract MSC describing the code shown in figure 4.5.

4.3.3 Distinguishing the Different Messages within a Component

Until now we have seen how we can extract a MSC from one single message within a component. A component can provide different services that are accessed by sending different messages. We need to make a distinction between these different messages. When considering a single component we do not know which other components will require the services of that

```
Component Dummy
{
  <other messages>

  message NotifyAll()
  {
    <some code>

    try
    {
      <Exception prone code>
    }
    catch (ExceptionType1 e)
    {
      <Type1ExceptionHandlingCode>
    }
    .
    .
    .
    catch (ExceptionTypeN e)
    {
      <TypeNExceptionHandlingCode>
    }

    <some code>
  }

  <other messages>
}
```

Figure 4.5: Example of Exceptions in Component-Code

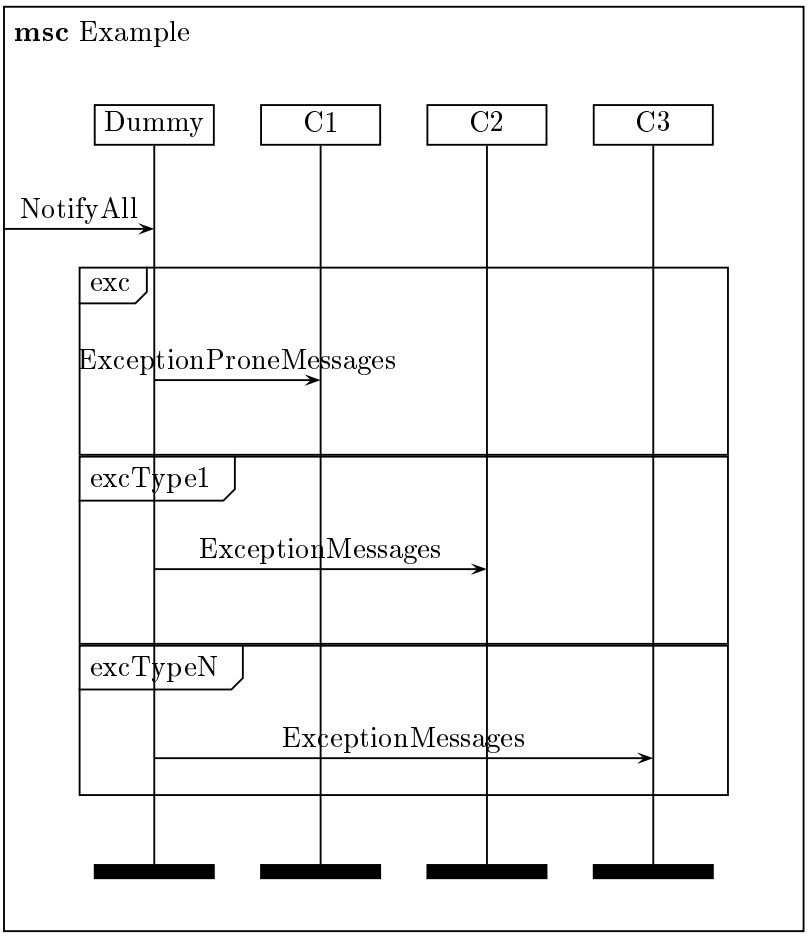

Figure 4.6: Abstract MSC describing exceptions

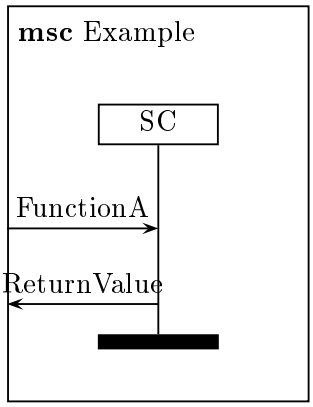

Figure 4.7: Example of a Return Message

component until they are all plugged together. Hence we cannot simply draw an arrow from all possible components that could send this message.

We will denote the receipt of a certain message by putting an arrow from the evironment to the component that implements the message, thus marking the sender of the message as "unknown". Figure 4.6 shows the description of the message *NotifyAll* provided by the component called Dummy.

4.3.4 Describing Return-Messages

In section 3.3.2 we explained that due to the usage of asynchronous communication we cannot just return the computed value of a function. Instead we have to send a message containing the result to the component that initiated the service. As we do not know the initiator of the message when considering the abstract behaviour we represent a return-message by an arrow to the environment. An example is shown in figure 4.7.

4.3.5 Undetermined Interface Providers

As it is not possible to determine the clients of a component it is not always possible to determine what components will be used to satisfy the required interfaces. Consider the piece of code in figure 4.8 of a message within a component. It is a classic example of a subscriber-dispatcher pattern [16] that can also be applied within components. It is clear that the receiver is undetermined here until the components are plugged together.

In [15] this is solved by putting different undetermined environments in the MSCs as component instances at that point. These environments are replaced by the actual component instances at composition-time. The disadvantage of this method is that at composition-time only one possible composition can be instantiated at a time. To capture the full behaviour of a

```
Component SC
{
  <other code>

  message NotifyAll()
  {
    for (int i=0; i<observers.size(); i++)
    {
     observers[i]..Update();
    }
  }

  <other code>
}
```

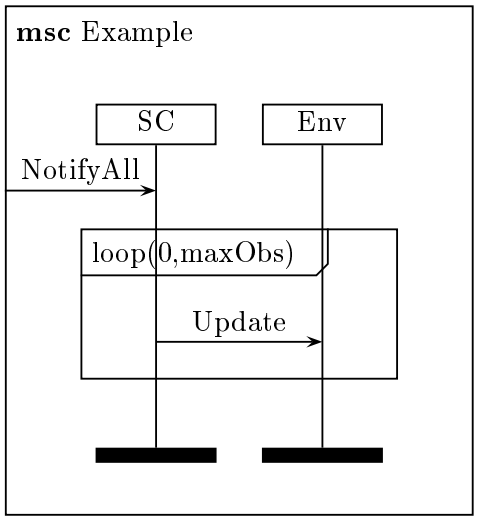

Figure 4.8: Undetermined receivers

program we need to consider all possible instances that are used at run-time. We will use this technique, but with the addition that at composition time all candidates are considered. This will be explained in the next chapter when we talk about the concrete communication behaviour.

4.4 Adding Message Parameters

Each message-send between components includes a set of parameters is included. To enhance the description of the code we will extend the abstract MSCs with the parameternames and their values. The parametername will be denoted on the arrow and the values are denoted after a double point. The value of the parameters is only known in some rare cases. If the value is unknown then we denote this with an asterisk. An example is shown in figure 4.9. Note that the component that is to receive the message *Update* is dynamically expressed using the parameter *Observer*. Parameters are important because they can be used to refine the possible execution traces in some situations. In chapter 9 we will also discuss why the inclusion of parameters is important to the scheduler.

4.5 Refining the MSCs using the Parameters

The code shown in figure 4.10 has a condition in the form of an if-then-else statement. The abstract MSC for this code is shown in figure 4.11. As we will see in chapter 9 it is sometimes usefull for the scheduler to know the difference between these alternatives. For this reason it is possible to define a refined MSC. A refined MSC displays a specific part of the general

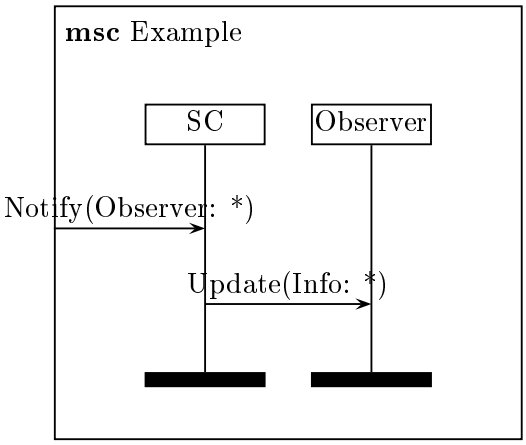

Figure 4.9: Denoting Parameters on an MSC

abstract MSC. A refined MSC that only describes what happens when the temperature is larger than the treshold is shown in figure 4.12. A boolean expression is placed in the condition to denote this specific case.

4.6 Summary

Message sequence charts can be used to describe a certain scenario between multiple communicating instances. They were initially intended for describing scenarios in telecommunication hardware and software. MSCs are not well suited for describing the behaviour of components. In this chapter we have discussed how we can use MSCs to document source code of the components. Further extensions to the MSCs are necessary to express the use of variables/unknowns in source code and the use of parameters in the communication. Finally we have also proposed a way to reduce the possible scenarios by refining the MSCs. Abstract MSCs are a first step towards the component documentation that is going to be used as input for the scheduler of the component system. This is also shown on figure 1.4 on page 10.

```
Component Watcher
{
  <other code>

  message NotifyAll()
  {
    int temperature = <Integer|Temperature>.intValue();
    if (temperature > TRESHOLD)
    {
      Cooler..Alert("Temperature", new Integer(temperature));
    }
    else
    {
      Reporter..Log("Temperature", new Integer(temperature));
    }
  }

  <other code>
}
```

Figure 4.10: Code from the Watcher Component

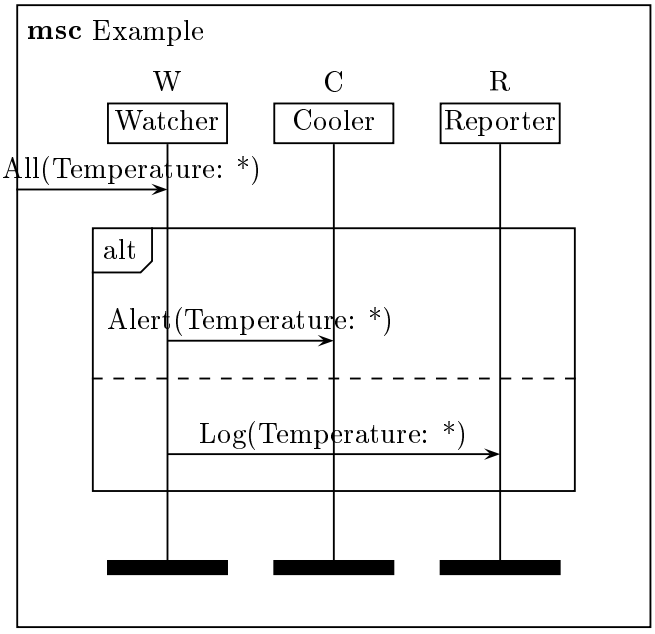

Figure 4.11: Abstract MSC for Watcher.NotifyAll()

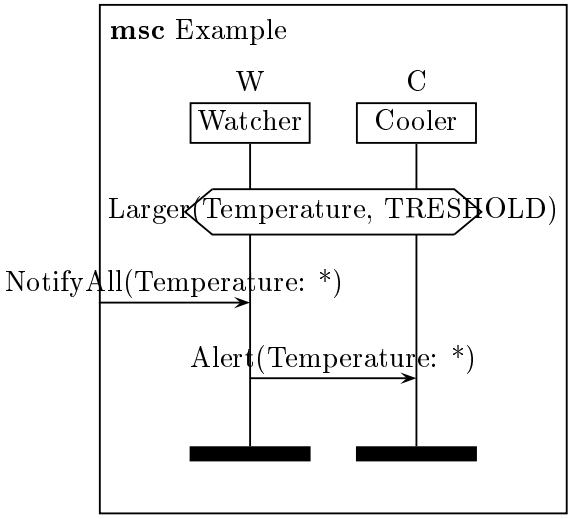

Figure 4.12: Refined MSC for Watcher.NotifyAll()

Chapter 5

Describing the Concrete Behaviour of a Program

The previous chapter discussed how we can describe the abstract behaviour from an abstract component. We have seen that we cannot construct the run-time communication behaviour with abstract MSCs. The run-time communication behaviour can only be constructed when the components are glued together. In the abstract MSCs we ended up with several variables. Most of these variables can be worked away by taking the abstract behaviour and making it concrete at component composition time. In chapter 2 we defined that each component instance has a unique name. In concrete MSCs these names are available and denoted above the box containing the type of the component.

5.1 Resolving unknowns

In the previous section we ended up with two problems when extracting the behaviour of a component from its source code:

1. We do not know what other components will be plugged in to provide the *required interfaces*

2. We do not know the *clients* of the component

5.1.1 Determining the Required Interface Providers

When the components are plugged together in a program, we can determine these variables. When we consider all the components used in that program, we know all their interfaces of the messages that are supported. Consider the example shown in figure 4.8. Imagine that when plugging the components together we have three component instances (named observer1, observer2 and observer3) that can respond to the *Update* message. Then we can

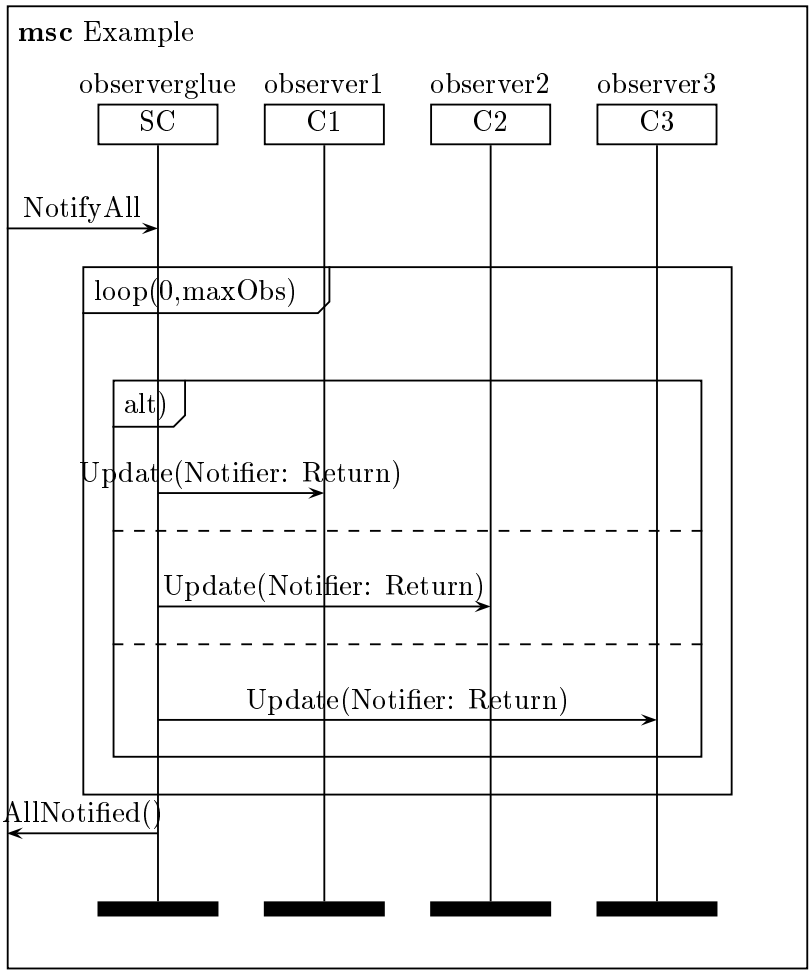

Figure 5.1: Solution to Undetermined Receivers

translate the MSC displayed in figure 4.8 to figure 5.1. We can repeat this process for all abstract MSCs with such unknowns. Figure 5.2 shows a more compact representation, by denoting the set of instances above the vertical line. A more compact representation is useful, because there might be a lot of possible components that support a certain interface.

5.1.2 Determining the Clients of a Component

After resolving the required interface providers we are left with a set of MSCs with one class of variables left to resolve, namely the component's clients. Consider the example shown in figure 5.1. We can walk through the set of MSCs and collect all the components that send the message *NotifyAll* to the component SC. Suppose that component C4 sends the *NotifyAll* message to

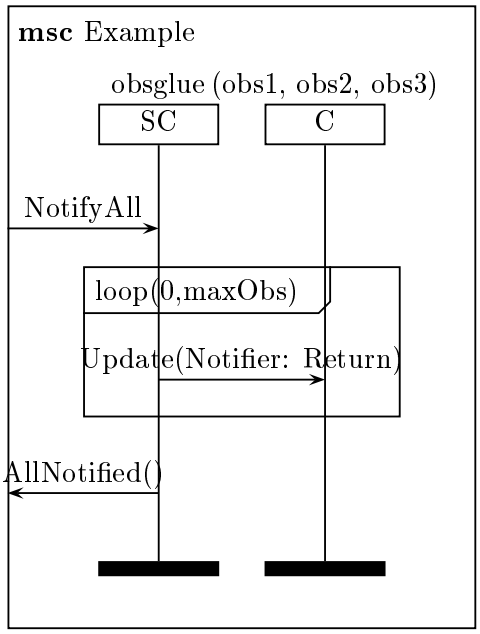

Figure 5.2: Compact Solution to Undetermined Receivers

SC then we replace the MSC on figure 5.1 with figure 5.3. We can use the same compact representation as used when determining the required interfaces to reduce the number of concrete MSCs.

5.2 Making Exceptions Concrete

In section 4.3.2 we have discussed how we can express try-catch statements in abstract MSCs. A typical use of exceptions is demonstrated in figure 4.5 on page 36. The semantic behaviour of this code is that when executing the exception-prone code a jump can be made to any of the exception-handling sections. Suppose that in exception-prone code, the messages *m1 ... mX* are sent, and in the exception handling section the message *mE* is sent. The possible execution traces of this code are shown in figure 5.4. If X messages are sent in the exception prone code and we have N exception-handling or catch statements then we have (X + 1) * N possible execution traces. In the concrete MSC all these possibilities are expressed using the alt inline operator expression. The above example shows that the use of exceptions together with component-communication should be used with care. In some cases it is possible to reduce the size of the concrete MSCs describing exception-handling code by limiting the code used in the try-block to the statements that can throw exceptions. This is not always possible, but a more compact representation for concrete MSCs without loss of important

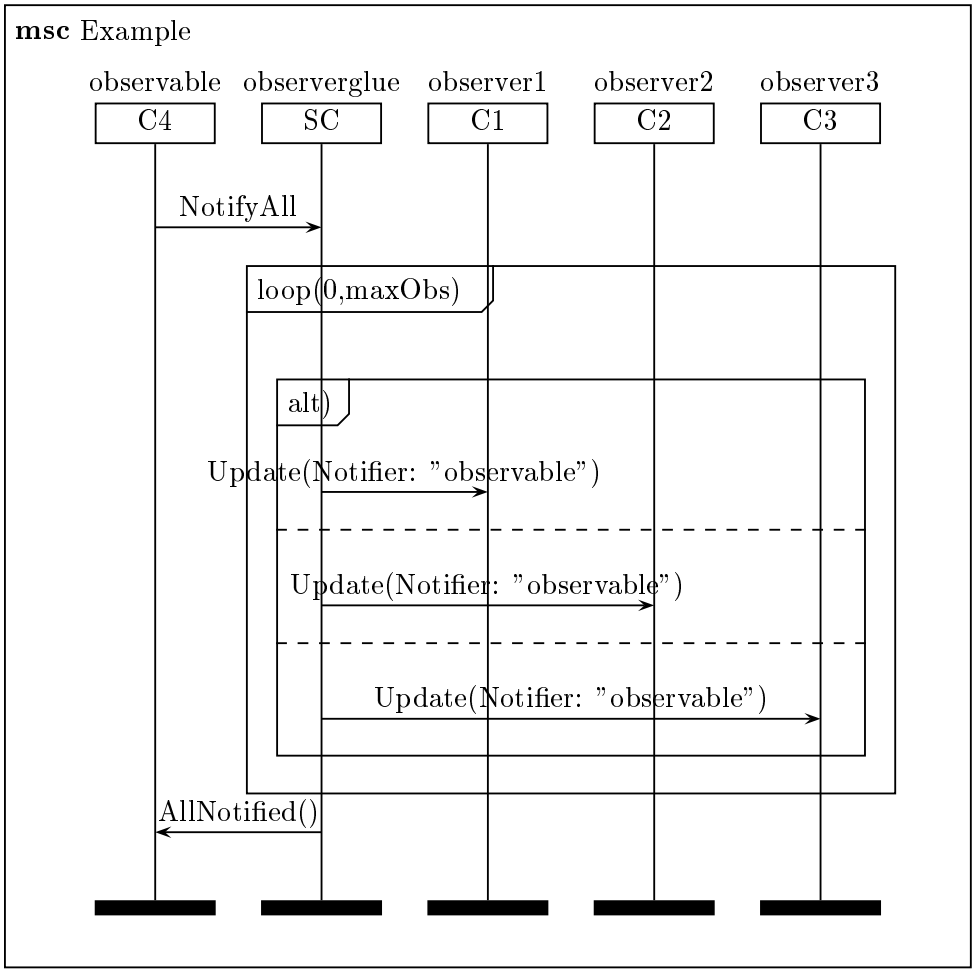

Figure 5.3: Solution to Undetermined Clients

```
0: mE
1: m1, mE
2: m1, m2, mE
.
.
.
X: m1, m2, ..., mX, mE
```

Figure 5.4: Possible execution traces

information is out of the scope if this dissertation.

5.3 Matching the Parameters

Sometimes we can derive the value of the parameters. This was the case in the example shown by figure 5.1 and figure 5.3. The parameter *Notifier* was first marked by the variable *Return*, but when the component instance observable was identified as a possible client the parameter *Notifier* was known. What we can do is propagate this variable to the concrete MSCs describing the behaviour of the *Update* message from the component instances observable1, observable2 and observable3. It is important that we do this, because the parameters can be used to refine the possible execution traces as described in section 4.5.

5.4 Summary

Concrete MSCs describe the run-time communication behaviour between the different components. They can only be extracted after the components have been plugged together into a system. It is possible to extract the concrete behaviour from the abstract MSCs. The extraction occurs in three different steps:

1. Resolving the unknowns from the abstract MSCs
2. Making the exceptions concrete
3. Matching the parameters used in the communication

When these three steps are completed we have a map of all run-time communication behaviour of the different components that are plugged together. The concrete MSCs are given as input to track the communication among the components. The track gives valuable feedback to the scheduler that is described in chapter 9. The concrete MSCs are part of the component documentation that is shown in figure 1.4 on page 10.

Chapter 6

Describing Real-Time Behaviour

In chapter 4 we have discussed how we can describe the abstract behaviour starting from a piece of component source code. Chapter 5 then discussed how we could extract the concrete behaviour of a program from the abstract behaviour. There are however some aspects, typical for real-time embedded software, that are not described in the abstract or concrete MSCs. These aspects are important to a scheduler and need to be described. This chapter discusses these aspects and how they can be described by means of MSCs.

6.1 Message Triggers

In chapter 1 we saw that one of the characteristics of embedded systems is that they are inextricably bound with their environment. One of the purposes of the software is to react to events that occur in that environment. We have two different ways in which embedded software can detect the events in its environment:

- When using *pushing detection* a hardware component will initiate an interrupt when an event is detected in its environment. This interrupt is then handled by the software.
- When using *polling detection*, the software polls the hardware for its status when it needs to know the latest changes.

Based on the way how changes in the environment are detected we can distinct two types of software:

1. Time-triggered software
2. Event-triggered software

Definition 14 (Time-triggered software) *Time-triggered software is software in which* all starting messages *are initiated by a timer at certain well-defined time intervals.*

Definition 15 (Event-triggered software) *Event-triggered software is software in which a starting message is initiated by an interrupt caused by one of the hardware components in the embedded system.*

6.1.1 Evaluation of time-triggered software vs. event-triggered software

Reliability

When we consider both approaches then the event-triggered software seems the most intuitive of both. When something changes in the environment then a message is sent and the software can react to that change. The drawback is that we cannot predict the behaviour of the environment. It is possible that within a certain time-frame a lot of events occur, which all cause messages to be sent so that the software can react to them. An upper bound on the maximum number of events that can occur in that timeframe can only be given when we have some foreknowledge from the environment, but sometimes it is simply impossible to give an upperbound. As the system will have to process all messages introduced into the system this can cause an overload. So event-triggered software greatly affects the predictability of a system, which is not what we want in a real-time device. In a system where all messages are time-triggered, the environment cannot influence the computational resources that are needed. In time-triggered software it is the software that controls the computational resources and not the environment. This characteristic makes time-triggered software more reliable than event-triggered software.

Resource Usage

A real-time device must be able to to detect certain events that occur in the environment within a certain time-frame. We can satisfy this requirement with time-triggered software by chosing a time-interval at which we check if the environment has changed. When we keep that time-interval small enough, guarantees can be made about the reaction time of the system to a certain event. The length of the interval must be small enough to cover all situations. This will cause the system to use computational resources even when no events occured. This results in the need for more resources, since other tasks still have to be processed in the meantime. An event-triggered system on the other hand will only process its data when an event occurs, so there is no computation needed to check the status of the hardware.

	Event-triggered	Time-triggered
Reliability	-	+
Resource Usage	+	-
Flexibility	+	-

Table 6.1: Comparison of Event-triggered vs. Time-triggered Software

Polyvalency

As mentioned above, event-triggered software only uses resources when events actually occur, while time-triggered software uses CPU power even when no events occurred. Event-triggered software can process more tasks with the same amount of resources making the system more polyvalent.

6.2 Describing Message Triggers

The component system described in chapter 3 supports both time-triggered and event-triggered messages. Both types of messages are interesting to describe, because they specify the introduction of requests to computation in our system. They are both expressed in the abstract or concrete MSCs, but can provide useful information to the scheduler as we will see in chapter 9.

We can distinguish three kinds of messages based on their frequency:

1. Aperiodic Messages
2. Periodic Messages
3. Sporadic Messages

We will now discuss their characteristics and how they can be described using message sequence charts.

6.2.1 Aperiodic Messages

An aperiodic message is a message which occurs unexpectedly. They can be compared to the event-triggered messages that were described above. Aperiodic messages will be denoted on abstract and concrete MSCs by putting a general condition on them. We can put a description in the condition, for example how the aperiodic message is initiated. This information will be of no use to the scheduler, but can promote the readability of the MSCs when they are being refined as described in section 4.5. The example shown in figure 6.1 describes that the component SC will send a message *Notify()* to the component RC if InterruptA occurs in the environment.

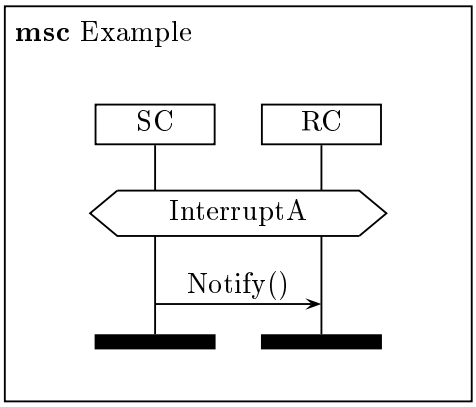

Figure 6.1: Example: Aperiodic Message

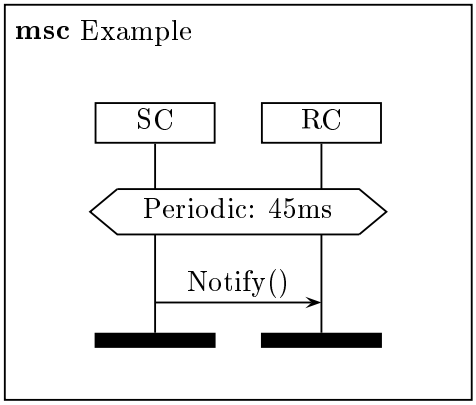

Figure 6.2: Example: Periodic Message

6.2.2 Periodic Messages

A periodic message is a message which recurs at a *regular* time interval. Periodic messages are denoted by a general condition similarly like aperiodic messages. The only difference is that periodic messages have a formal description in their condition. This formal description is denoted by placing the keyword *Periodic* followed by a colon and a number which denotes the recurrence rate. The example shown in figure 6.2 describes that the component SC sends the message *Notify()* every 45 milliseconds to the component RC.

6.2.3 Sporadic Messages

A sporadic message is a message which is recurrent, but *not regular*. There is however a minimum interarrival time between the messages that are send.

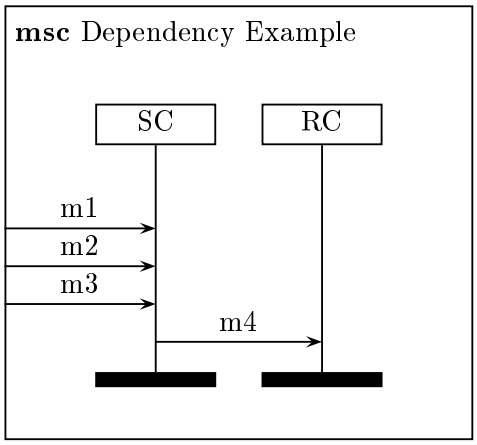

Figure 6.3: Example: Describing Dependencies

[19] shows that sporadic messages can be considered as periodic messages. Sporadic messages are denoted similarly as a periodic message, but the keyword *Periodic* is replaced by *Sporadic*.

6.3 Dependencies

In the text above we have always assumed that when a message is processed by a component, it requested services from another component or requested no services anymore. This is not always the case, sometimes a component is waiting for several different messages before it will send a message. Dependencies must be explicitly described. When a deadline must be met and a component is waiting for other messages, the scheduler needs to know what components the system is waiting for. The use of dependencies in a scheduler is discussed in chapter 9. Figure 6.3 shows that component SC is waiting for messages *m1, m2 and m3* before it sends *m4* to component DC. Note that this is an *abstract MSC* as we do not know what components will send *m1, m2 and m3*. We can extract a concrete MSC at component composition-time in a similar way as described in section 5.1.2.

6.4 Specifying Time Constraints

Until now, two different kinds of MSCs have been introduced to describe the behaviour of components and the behaviour of the components at run-time, namely *abstract* and *concrete* MSCs. All this information can be used by a scheduler to make some decisions. There is however a third kind of MSC that needs to be introduced. We need to *specify* the timing behaviour that is expected from the software when the components are plugged together.

In this section we shortly discuss the syntax of these three specification methods. Finally we explain what specification syntax we use for timing constraints throughout the remainder of this dissertation.

6.4.1 Existing Syntax

To facilitate the specification of real-time systems with MSCs, a few extensions have been proposed [11, 12, 13] to express timing constraints:

- timers
- delay intervals
- timing markers

Timers

Recommendation Z.120 [11] provides *timers* to express timing constraints in a basic MSC. Within a single instance a timer can be placed on the basic MSC. It is important to note that this timer cannot be shared among different instances. We can perform three functions on a timer. A timer can be:

1. *set* to a value
2. *reset* to zero
3. *observed* for timeout

Figure 6.4 shows an example of three components sending messages amongst each other. Timer T3 expresses that it is set to some value, say 7 time units and that it sends m2, receives m3 and sends m4 *before* T3 is observed for timeout. Timer T1 expresses that it is set to some value, say 5 time units and after it has received message m1 the timer is reset. The implicit assumption is made that the timer is reset *after* it has expired. A timer can be used to express two different timing constraints:

1. a *maximal* delay is expressed by timer T3, it actually denotes that one component must exchange some messages within a given time bound.
2. a *minimal* delay is expressed by timer T1. T1 expresses that receiving message m1 *at least* takes a certain time.

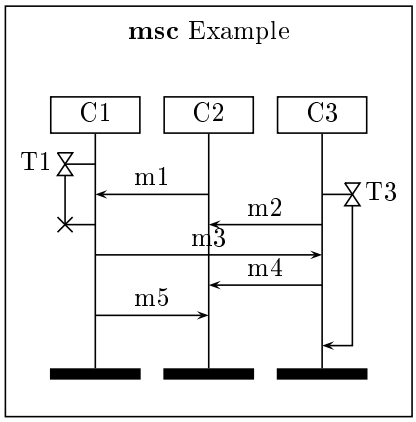

Figure 6.4: Example of a basic MSC with timers

Delay Intervals

Delay intervals is another method that has been proposed by [12, 13] to express timing constraints on basic MSCs. Delay intervals can be used to express three types of timing constraints:

1. *Event-associated* timing constraints are expressed as an interval next to an event (message sent or received).

2. *Message delivery* delays are denoted as a delay over the message arrows.

3. *Processor's speed* constraints are expressed as an interval between two consecutive events in a component.

Event-associated timing constraints are constraints that are expressed on a basic MSC. They denote *global* timing constraints on events, namely a message send or message receipt: the event must occur within the minimal and maximal time delays with respect to *any* previous event, whenever it occurs in a trace. An example is shown is figure 6.5.

Message delivery delays and processor's speed constraints are intervals that are put between two visually ordered message sends. Figure 6.6 shows an example of message delivery delays, mixed with processor's speed constraints. It specifies that m4 will be processed by C2 between at least 1 time unit and at most 3 time units after it has been send by C3. It also expresses that C3 will process m3 between one and two time units after it has sent m2.

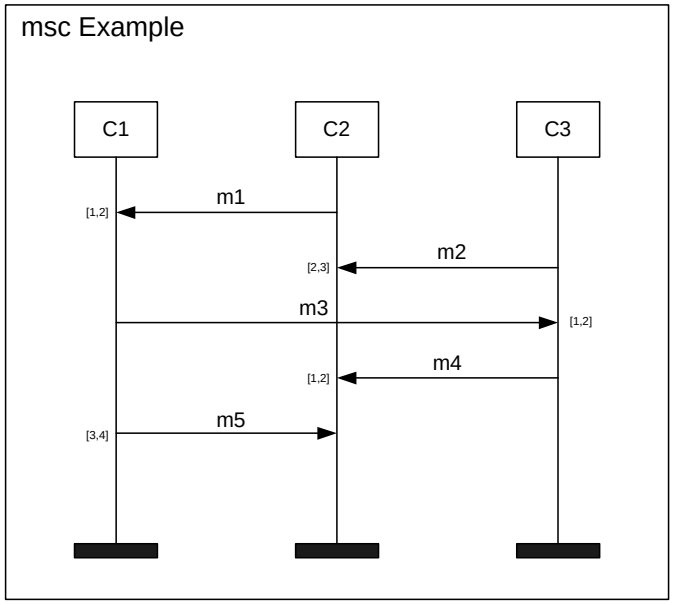

Figure 6.5: Example of Event-associated Timing Constraints

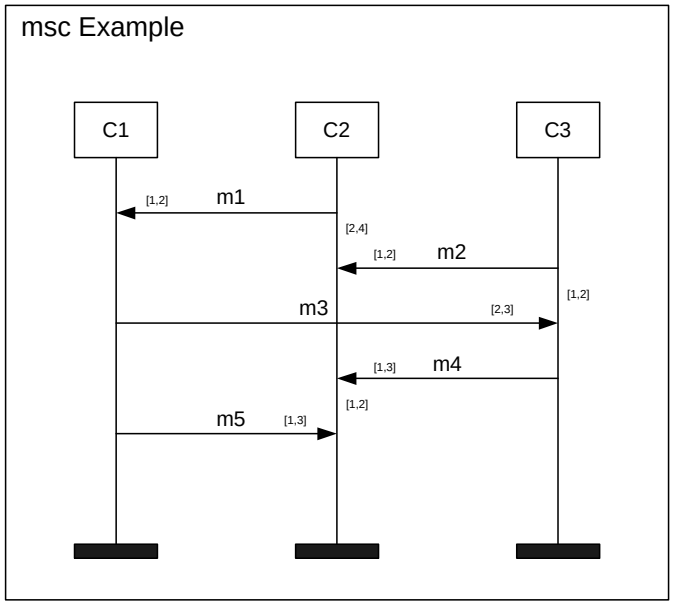

Figure 6.6: Example of Delivery Delays and Processor's Speed Constraints

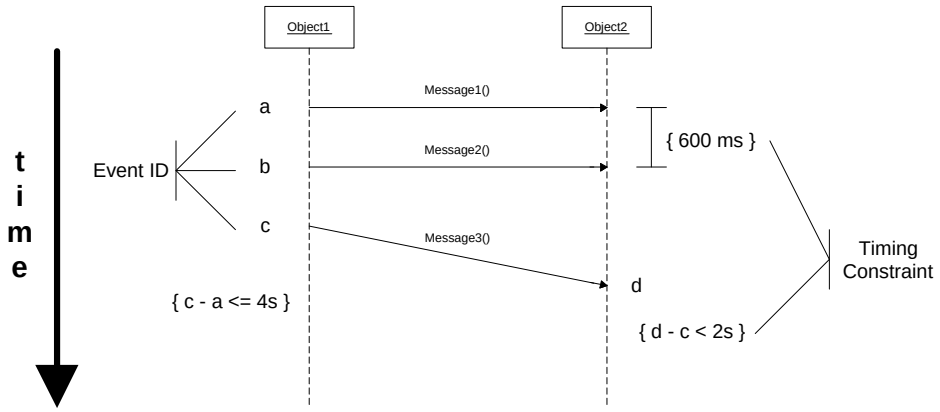

Figure 6.7: Example of Time Constraints on UML Sequence Diagrams

Timing Markers

In the Real-Time Unified Modeling Language [17](RT-UML) the syntax of the different UML notations has been extended to express real-time constraints. Sequence diagrams are used within UML to describe object-interaction scenarios and have been extended to express timing constraints. Sequence diagrams are very similar to MSCs and their notation could be used to express timing constraints on MSCs as well. A straight line to an object means that the message is immediately processed by the other object (i.e. a synchronous message). When an arrow is slated downwards, a delay is expressed between the time of sending and processing of the message. An identifier can be given to an event on sequence diagram. Using the event-identifiers, it is possible to formulate timing constraints on the sequence diagram. An example is shown in figure 6.7. In this example, the constraint

```
d - c < 2s
```

denotes that object2 must have processed message3() at most two seconds after it has been sent by object1. Another time-constraint

```
c - a <= 4s
```

expresses that the time between the occurence of message3 and message1 must be less than or equal to 4 seconds.

6.4.2 Review of the Different Methods

When a program is composed using components we want to specify the temporal behaviour of the software. Typically when a message m is sent

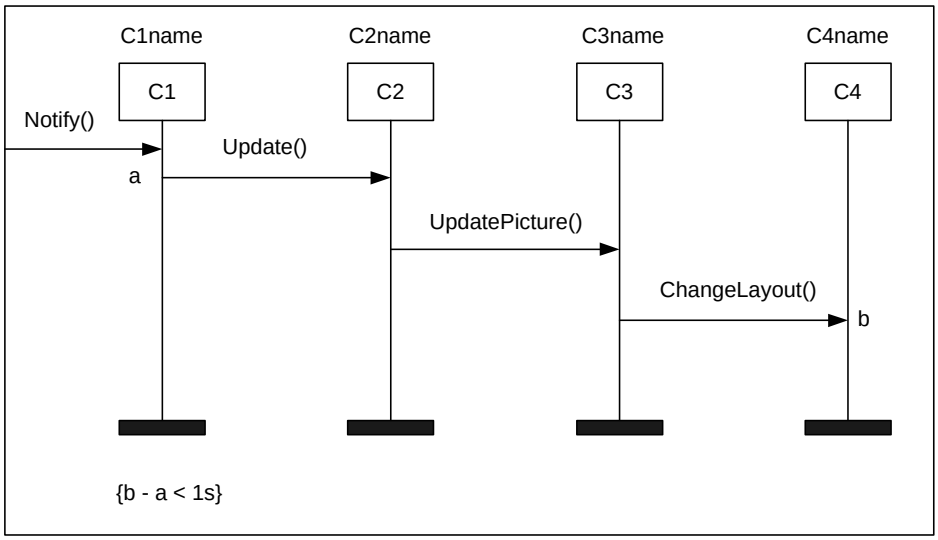

Figure 6.8: Example of an MSC with timing markers

to component X, this results in the sending of different other messages requesting services from other components. What we want to express is a timing constraint on a chain of messages, because the different services are distributed over different components. Such timing constraints cannot be expressed using timers as they are local to the instance. Delay intervals could be used to specify the timing constraints distributed over the different instances by specifying message delivery delays on each message that is sent. One drawback of these message delivery delays is that they require that each message has a deadline. Message delivery delays do not allow you to specify one timing constraint over the whole chain. Timing markers allow you to express time constraints over a chain of messages. To improve the expressibility of timers and delay intervals they could be used together [14]. In [14] an algorithm is also described to detect inconsistencies on an MSC using timers and delay intervals. Timers and delay intervals are better at describing timing constraints that are local to the instances. We use the timing markers on MSCs instead of sequence diagrams (an example is shown in figure 6.8). One difference however, is that a message will always have a straight arrow, even when the message is not processed immediately.

6.5 Summary

In the first part of this chapter we have seen how two shortcomings of message sequence charts have been resolved, namely the introduction of requests for computation and dependencies. The introduction of requests for compu-

tation can occur with event-triggered messages and time-triggered messages. Event-triggered messages are introduced into the system by hardware interrupts. Time-triggered messages are messages that recur in regular time intervals. With time-triggered messages the software has full control over the computational resources that are used, whereas event-triggered messages are dependent on the environment. Systems with event-triggered messages have a greater flexibility, because they use less resources than software using time-triggered messages. Dependent messages are messages which are dependent on the receival of several other messages. They are interesting to document, because they can provide the scheduler with information that allows it to speed up the dependent parts when waiting for such a message. A third class of documentation is necessary to specify the required temporal behaviour of a certain program. This class of MSCs are called timing-marked MSCs which allow the developer to define timing constraints over a chain of communication messages between the different components. Timing-marked MSCs are regular MSCs with a mark on certain events. The expression of the timing constraint is denoted with a simple Boolean expression. This third class of MSCs completes the component documentation shown in figure 1.4 on page 10. They are used as input by the tracker (chapter 8) and the scheduler (chapter 9) to enforce the specified timing constraints. They also play an important role in the reduction of the overhead by limiting the number of threads, which is discussed in chapter 11.

Part II

Scheduling

Chapter 7

Schedulability Analysis

In part 1 we have seen how we could document software components by using abstract MSCs. The run-time communication behaviour of the components could then be extracted by converting the abstract MSCs to concrete MSCs, when the components are glued together. In chapter 6 we discussed how you can specify real-time constraints on the concrete behaviour. Hence it would be interesting if we could check if the hardware that is to be used in the embedded system will provide enough resources so that the specified time constraints can be met. This is an important phase in the development stage. Electrical engineers have to make ad-hoc decisions on the hardware that is to be used. An overdimensioned system causes the production costs of the embedded systems to be too high, which is economically not feasible. A prototype with not enough resources causes the need to redesign a new prototype with more resources available. This need for a redesign can cause the production cycle to be delayed having economical implications as well. In the remainder of this chapter we will explain how such an analysis can be done, but first we will discuss how we can capture the execution times of the different components.

7.1 Execution Time

In an asynchronous communication model we have three different notions of time we need to consider.

Definition 16 (Transmission Time) *The transmission time of message m relative to message k (denoted by the function TT(k, m)) is the time it takes to send the message m when message k is being processed by the receiving component.*

Definition 17 (Delivery Time) *The delivery time of message m (denoted by DT(m)) is the time interval between when the message m was sent and when the component starts to process message m.*

Definition 18 (Execution Time) *The execution time of message m by component C (denoted by ET(C,m)) is the time it takes for component C to execute the code bound to the message m.*

The three notions of time are related to the speed of the hardware from the embedded system and to the state of the system.

We can distinguish at least three different execution times:

1. *Best-case (minimal) execution time* is the best possible execution time for a piece of code.

2. *Average execution time* is the average execution time over a number of different runs of the system.

3. *Worst-case (maximal) execution time* is the longest possible execution time a piece of code can have.

It is important to realize that the use of these different execution times in a schedulability analysis leads to different systems. In a mission-critical hard real-time system we will choose to use the worst-case execution times for the schedulability analysis, because the system cannot miss any deadline. A system with soft real-time deadlines on the other hand can make use of the average execution times, so that the embedded system keeps most of its deadlines.

7.1.1 Problems with Measuring Execution Time

Finding the execution time of software involves many problems and different programming constructs must be taken into account. Below is a summary of the problems we can encounter and their implications on the execution times. You may already be familiar with many of the problems, but they are mentioned here anyway to draw your attention on the level of difficulty involved in measuring the execution time.

Branches (such as if-statements, switches, etc...) can influence the execution flow and therefore the execution time of a program. When measuring execution times all branches should be considered.

Iteration and recursion are another aspect that influence the execution time. In worst case the iteration does not stop (infinite loops) which results in an infinite execution time. In best case the execution time is not prolonged when the loop is not entered. Infinite loops are used a lot in embedded software to continuously check for some change in the environment of the embedded system. We come back to the topic of loops in section 7.2.

The length of the datastructures causes different algorithms to exhibit different execution times. It should be taken into account that some datastructures grow and shrink dynamically over time. The algorithms that are used to handle these datastructures can influence the execution time immensely (e.g. quicksort vs. bubblesort).

Late binding in object-oriented languages causes the execution time to depend on the run-time properties instead of the static properties of on the code. When a new object is introduced into the system, the timing behaviour of the software can change completely.

Network connections provide another source for variation in the execution time. When an embedded system is hooked up to a network it is important to realize that the time for sending and receiving data over a network depends on the load of the network and/or the load of the other device used in the communication.

Concurrency is something that is ubiquitous in most embedded systems. When running different processes on a single CPU, the CPU time is multiplexed over these processes. It is obvious, but easily forgotten, that the execution time of the software depends on how many processes the CPU is currently serving.

Reusability When using component-based software in different embedded systems we need to deal with some problems that are bound to the reusability of the components. A nave solution could be that the component developer adds the execution times of the component to the documentation. The problem with this solution is that when the component is reused on other hardware all execution times delivered in the documentation are useless, unless we have an hardware-independent representation for execution times (below we will see such a representation).

7.1.2 Expressing the Execution Time

When we measure the execution times of the code we have to decide what units we will use to denote the results. There are basically two ways to express the execution time:

1. Time

2. Processor Ticks

Regular Time is maybe the most natural way to express the execution time. We can choose the granularity (e.g. milliseconds) according to the needs of the application. The problem with expressing the duration of the execution of a piece of code is that it is dependent on the hardware on which the test were done.

Processor Ticks are another way to denote the duration of a piece of code. Processor ticks are particularly useful in expressing interpreted execution times. E.g., we can alter the byte-code interpreter of a Virtual Machine to count the number of interpreted instructions. Such processor ticks are independent of the processor that is used. When given a processor we can do tests on the execution time of the byte-code on that processor and use these measurements to convert the processor ticks to regular time. The advantage is that the time expressed in processor ticks can be converted easily to be expressed in regular time given a certain processor type P:

$$TotalExecutionTime = ProcessorTicks * BytecodeInterpretTimeonP$$

A disadvantage is that the applicability is limited to interpreted code. Another disadvantage is that processor ticks converted to regular time will not be as accurate as regular time that was measured. This is because in one processor tick, some code is interpreted and evaluating the interpreted code varies on what is interpreted. This variation needs to be resolved by using statistics which introduces some variance in the measured results.

7.1.3 Test Sets

To solve some of the problems measuring the execution times (see section 7.1.1) we suggest to create a test set. The test set can be used to alter the state of the component (e.g. the worst-case execution state). This can be achieved by changing the internal data-structures of the component. To be able to do this the test set must have access to the internals of the component. Once the component's state is altered it is possible to measure the execution times. E.g., suppose we have a phonebook component that keeps the names and telephone numbers in a vector. With a test set we fill the vector to its maximum capacity with unsorted sample data. After having done that we can find the worst-case execution time of the lookup message by searching for the last element in the vector.

Creating a test set is not an easy task and is still error-prone, but once it is created it can be used and reused without breaking the black-box principles that are associated with components. Once a test set is created we can use *processor ticks* or *regular time* as our unit of measurement. The choice between these two has some implications.

Comparing both Strategies

When we use regular time as our time unit we will have to deliver the test sets with the components as documentation. This is necessary, because the results will need to be reproduced in different hardware environments. When we choose for processor ticks as time unit then the results are less accurate, but the test set is not necessary and the time can immediately be calculated for a certain processor. An advantage of using processor ticks is that a time estimate can be made before an actual prototype is built. We propose that when both units of measurements can be applied (since processor ticks can only be used in interpreted code) we deliver the components with both a test set and the processor ticks. This way a possible customer of a certain client can use the processor ticks as an indication before buying the component. When the customer has bought a certain component, more accurate results can be produced using the test sets.

Advantages and Difficulties of Creating a Test Set

Creating a test set is a complex task that should not be underestimated. When creating a test set we have to take most of the difficulties into account that were discussed in section 7.1.1. However, the creation of a test set also has different advantages:

- Reusability of components is not compromised using this method
- It becomes easy to switch to other hardware environments
- Writing a test set is a good way of writing correct code, which adds to the robustness of the system.
- When a test set is constructed, the developer will have to reason over the internal structure and algorithms used in the components, which can unveil flaws in the code.

7.2 Problem: Loops

Loops can create three kinds of troubling situations in our system:

1. When an infinite loop is used in a component, the component is not able to process the other messages in its queue.
2. If intercomponent communication occurs in a loop then the queue could become overloaded, which is a threat to the responsiveness of the system.
3. In section 2.3 we saw that some components need services from other components before they can deliver their services. A loop can occur when some components request each other's service.

In the component system we encounter two kinds of loops: *loops embedded in code*, which cover the first two situations and *loops due to inter-component communication* which covers the last situation. Both will be discussed now.

7.2.1 Loops Embedded in Code

Loop structures are omnipresent in programming languages (e.g. for, while, ...), but can also be hidden in recursions and other constructs (e.g. method 'a calls method 'b, and method 'b calls method 'a; a more complicated example: when a method 'a in a subclass calls the super method 'b, which in turn calls method 'a on self). We will make a distinction between infinite loops and finite loops.

Infinite Loops

Infinite loops are not allowed within a component, because the component system does not allow a component instance to handle more than one request at a time. A component cannot respond to incoming requests when an infinite loop is entered to process a message. As infinite loops are not allowed in the component system one might wonder if they can be avoided at all times. Consider a system that needs to poll a certain hardware device for a change in the environment. A system like this is regularly implemented as an infinite loop that polls the hardware device at each turn. In cases like these a time-triggered event (see section 6.1) should be used instead of an infinite loop.

Finite Loops

Whenever a loop embedded in code sends messages to other components, these messages are added to the queue. Since we cannot change the order of the incoming messages we have to process all these messages before the component can process another message. This way a finite loop can also influence the responsiveness of the system. Finite loops are sometimes unavoidable in code (e.g. when they are used to send a notification to a number of components in the observer-pattern). Instead of banning loops, the component manufacturer must add a lower- and upperbound on loops that contains intercomponent communication and influence components involved in timing constraints. The upper- and lower bounds are expressed in the inline operator expression on the abstract MSC. An example is shown in figure 7.1. This extra documentation is needed so that the scheduler can take appropriate actions or when a schedulability analysis (see also section 7.3) is done. Note that loops that do not interact with components involved in time constraints do not need to be documented.

```
Component SC
{
  <other code>

  message Approximate()
  {
    while (NewValue>delta)
    {
     <some computation>
     Target..Update(<NewValue>);
    }
  }

  <other code>
}
```

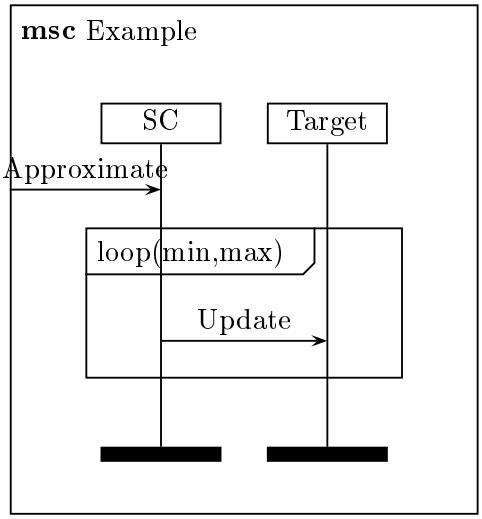

Figure 7.1: Documented Loops

7.2.2 Loops Encountered in Inter-Component Communication

Besides loops that are embedded in the code, we can also encounter loops in the intercomponent communication. Consider the following scenario:

1. Component A processes m0 and sends a message m1 to component B
2. Component B processes m1 and sends a message m2 to component C
3. Component C processes m2 and sends a message m3 to component D
4. Component D processes m3 and sends a message m0 to component A

This is an example of a loop in the intercomponent communication. To see the effect of such loops on the the reliability of the system we need to get into the internal structure of the component system. When an intercomponent message is sent, that message is placed in some queue for further execution. At some point in time that message is then processed and sends another message which ends up in a queue. The queue structure is pictured in figure 7.2. This picture shows that other components can still send messages to the components A, B, C and D, while that loop is occuring. Although the response time to incoming messages will be prolonged, the components will still be able to respond to them (incoming messages are still enqueued). Loops in intercomponent communication do not block the system in constrast to loops embedded in the code. Another problem we have when dealing with intercomponent communication is that it is impossible to know when the loop will be finished. There is a possibility that the loop will occur several times and after that a single message is sent with

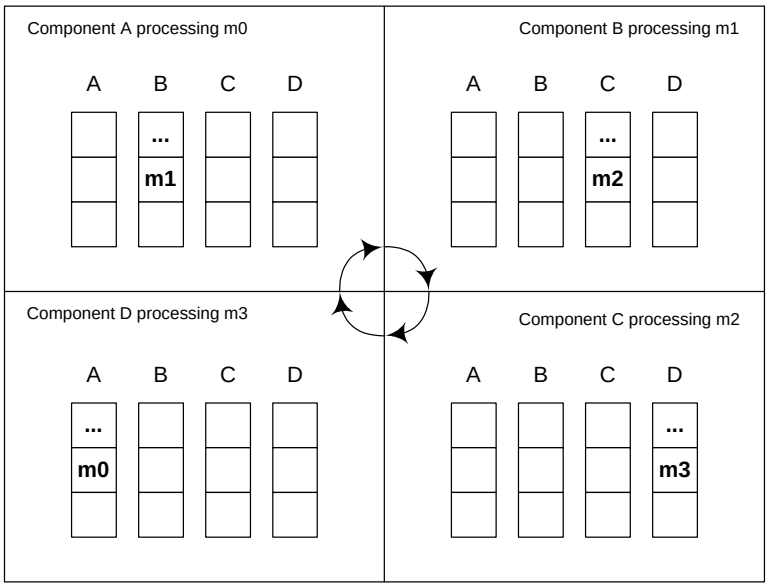

Figure 7.2: Intercomponent Loops - Internal Queues

some result. In that case it would be nice to know the maximum number of loops that are required before this message is sent. This information could be used by the scheduler or to do a schedulability analysis. Loops in intercomponent communication can be documented by placing an upper- and lowerbound in the inline operator expression. The example described above is shown in figure 7.3. Similar to loops embedded in code that send messages, there is no need to document these loops when they do not involve components that deliver real-time services.

7.3 Schedulability Test

[19] distinguishes two levels of accuracy in schedulability analyses:

Definition 19 (Necessary Schedulability Test) *If a necessary schedulability test is positive, the system will definitely satisfy the specified timing constraints.*

Definition 20 (Sufficient Schedulability Test) *If a sufficient schedulability test is negative, the system cannot satisfy the specified timing constraints.*

In the remainder we explain how a *Sufficient Schedulability Analysis* can be implemented, given the execution times and a timing constraint. Consider the MSC shown in figure 6.8 on page 58 and the average execution times shown in table 7.1. It is not difficult to see now that

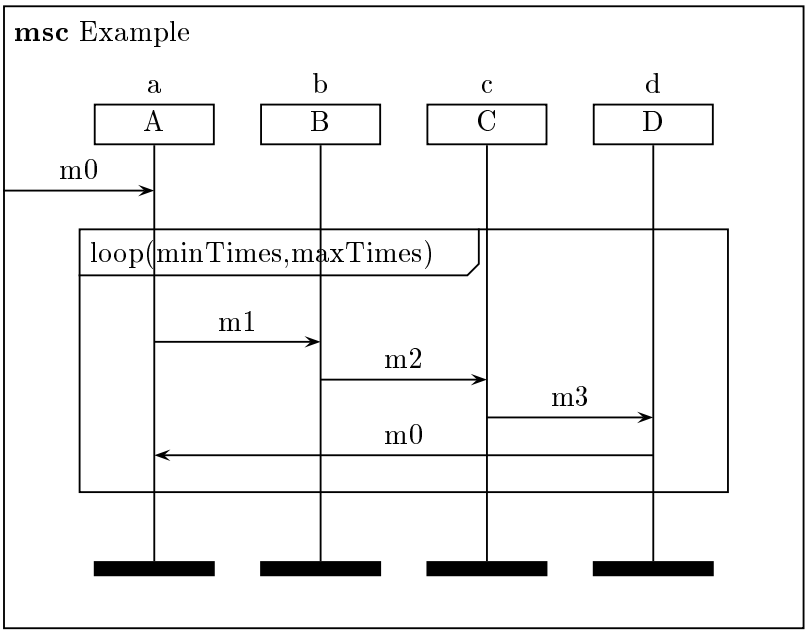

Figure 7.3: Documented Intercomponent Loops

```
234ms + 500ms + 900ms > 1s
```

and therefore the timing constraint cannot be met. When a loop is used in a timing-marked MSC we can calculate this by multiplying the execution time of the messages that are send within the loop with the maximum number of iterations of that loop. The reason why this is a *Sufficient Schedulability Analysis* comes from the fact that we do not consider the *delivery time*. Finding the delivery time is actually far more complicated. The delivery time in the component system is relative to the length of the internal queue. If we can find an upper-bound length of each queue in the system, then we can do a *Necessary Schedulability Analysis*.

Let us assume that we have an upper-bound length for the queues of each component (denoted by MQL(C)) shown in table 7.2. We also need to know for each component what message has the highest worst-case execution time (denoted by WCET(C)). These execution times are also denoted in that table. The necessary schedulability test for our example is shown in figure 7.4. In a hard real-time embedded system a necessary schedulability test is indispensable, but because the emphasis of this dissertation is not schedulability analysis we have not tried to find such an upper-bound. If the component system is to be used in a hard real-time system a formal proof should be given for an upperbound length of each queue.

```
(MQL(C2) * WCET(C2) + ET(C2, Update)) +
(MQL(C3) * WCET(C3) + ET(C3, UpdatePicture)) +
(MQL(C4) * WCET(C4) + ET(C4, UpdateLayout))} < 1s

MQL  = Maximum Queue Length
WCET = WorstCase Execution Time
```

Figure 7.4: Necessary Schedulability Test

Component Name	Message Name	Execution Time
C2	Update()	234ms
C3	UpdatePicture()	500ms
C4	UpdateLayout()	900ms

Table 7.1: Sample List of the Average Execution Times

Component Name	Max Queue Length MQL(C)	Worst Case Execution Time WCET(C)
C2	23	654ms
C3	43	543ms
C4	12	674ms

Table 7.2: Extra Information Needed for a Sufficient Schedulability Test

7.4 Summary

A schedulability analysis can be useful to predict whether the hardware used in the embedded system will provide enough computational resources for its task. The execution times of the different tasks are important to do such a schedulability analysis. Finding the execution time is not an obvious task, which involves careful analysis of the source code. We need a transparent way to provide the execution time as documentation with the components. We propose to create a test set, which can revert the component into a certain state. Once the component is reverted into a certain state tests can be done to measure the execution time of the different messages. Another choice that has to be made is the unit of time. We considered *regular time* and the number of *processor ticks*. Processor ticks can be converted to regular time. Processor ticks have the advantage that they are independent of the hardware used, but they can only be used in interpreted code. Another disadvantage is that the conversion of processor ticks to regular time is not entirely accurate. An advantage of the processor ticks is that they provide an estimate of the regular time *before* the embedded device is actually build. This way the electrical engineers can make an educated guess of the hardware that is needed in the device they are building. With the timing-marked MSCs and the timing information of the different components we can perform a sufficient schedulability test. A necessary schedulability test is however necessary if we want to compose hard real-time systems. To do a necessary schedulability test we need an upperbound on the length of queues from the different components.

Chapter 8

Tracking the Execution of a Program

In the remainder of this dissertation a scheme will be presented that adjusts the processing rate of the different components. The processing rate will be changed according to the execution state of the different components and the timing constraints that were specified using the techniques specified in chapter 6. This means we need to extend the component system with a mechanism that records the current execution state of the running program. In this chapter we will present the techniques that were used to record the execution trace of the program. The technique presented here tracks the component communication to follow the current execution trace of the program. Tracing the execution occurs on two levels:

1. Message-based tracking
2. Constraint-based tracking

8.1 Message-Based Tracking

A *message-based tracker* follows the execution trace of one specific message within a component. So for each component in a program, there is another tracker. In this section we discuss the internals of a message-based tracker (MBT). When plugging the different components together the concrete behaviour can be constructed using the technique explained in chapter 5. The concrete behaviour describes all intercomponent communication. A MBT uses the concrete behaviour of its component to track the execution trace of that component's intercomponent communication at run-time. To simplify tracking the execution trace a conversion of the concrete behaviour to a non-deterministic finite automate (NFA) can be done. Figure 8.2 shows the NFA corresponding to the concrete MSC shown in figure 8.1. A constructive proof by induction is given in [20], which shows that from each NFA a DFA

can be constructed that has the same language of the NFA. The NFA is therefore converted to a DFA using the algorithm defined in the constructive proof. When *NotifyAll* is processed by component "Dispatcher" and the component sends a message *Update()* to the *Log* component then the following happens:

1. initially we are in the start-state
2. we match the message *Log..Update()* to all outgoing transitions (this can also include matching the parameters that are passed along with the message)
   (a) if it matches the transition's destination becomes the current state
   (b) if it does not match we try another transition
3. eventually we move to the new state corresponding to the transition that was found
4. if we are not in an end-state we go back to 2.

This algorithm is good if we have exactly one DFA / message in a component. In section 4.5 I have explained that we can refine the communication traces by adding a boolean value and creating a more specific abstract MSC. This means we can have more than one abstract MSC describing the behaviour of a message, which implies we can end up with different DFAs to match. We can combine the different DFAs into a NFA. After doing this we can convert the NFA back to one DFA. We can get rid of the run-time overhead created by converting the NFAs to DFAs by converting the NFAs at component composition time.

The overhead of matching the different transitions can be minimised by using POOL-structures. In a pool structure each message has a unique identifier. Matching the message over the transition can be done in O(1) if a lookup table is constructed at compile-time. A similar POOL-structure could be used for matching the parameters, but this can only be done if the component manufacturer has foreseen this in the parameter objects. The overhead of parameter matching can be reduced by using less refined abstract MSCs, but the price is that the scheduler will be less accurate. We come back to this topic in chapter 9 about Scheduling.

8.2 Constraint-Based Tracking

Besides the tracking on message level, the scheduler needs information about the progress of the different timing constraints to take some decisions. Tracking the progress of the timing constraints is called constraint-based tracking

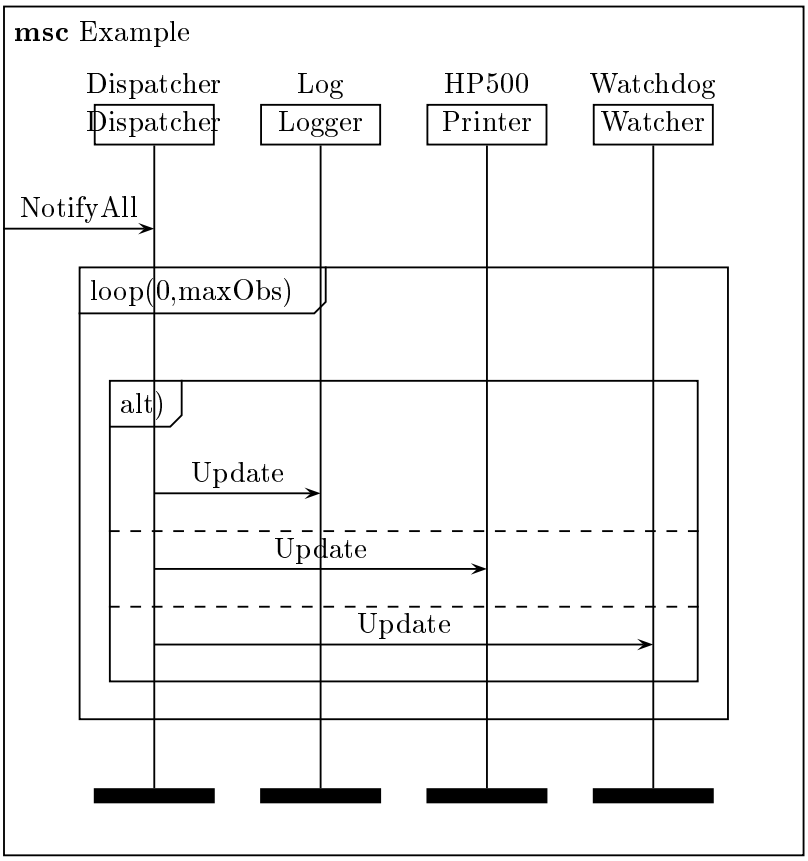

Figure 8.1: Example of a Concrete MSC

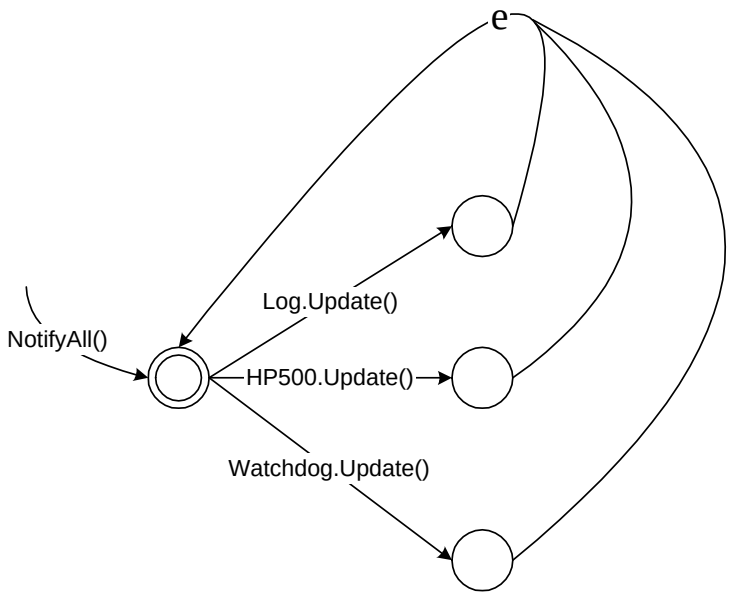

Figure 8.2: NFA corresponding to the Concrete MSC

(CBT). In this section we show how the time constraints can efficiently be tracked in O(1). The CBT is separated in two different activities:

1. Detection of a starting constraint

2. Tracking the progress of a constraint

Detection of a starting constraint In the previous section we have seen how the concrete MSCs can be converted into a DFA. The input of the algorithm is a timing-marked MSC (see section 6.4.2) and the set of DFAs that we constructed from the concrete behaviour. When we consider the example shown in figure 6.8 on page 58 then we can see the timing constraint is initiated by sending a message Notify() to the component C1name. The algorithm presented below adapts the DFAs so that they can help in identifying the timing constraints:

1. for each DFA d

   (a) for each transition t

      i. if the transition t matches with the timing-marked DFA

      ii. then add the timing-marked DFA to the transition information

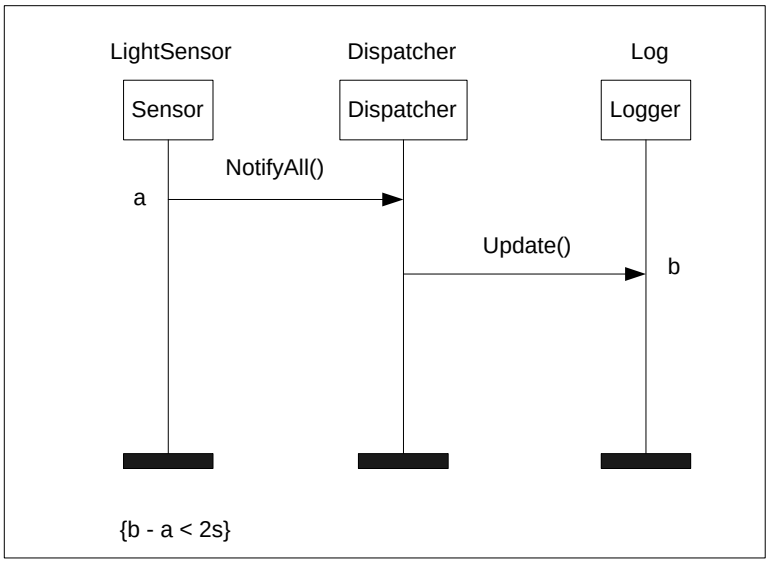

Figure 8.3: Timing-marked MSC for the Logger

If we do this for each timing-marked DFA that we have specified then all the DFAs carry all starting points of the timing-marked MSCs. Note that this operation can be carried out at compile-time creating no run-time overhead. When at run-time a transition is matched that contains a reference to a timing-marked DFA then two things are done:

1. a unique identifier is placed as a hidden parameter in that message
2. the DFA of the timing-marked MSC is placed in a lookup table using the unique identifier

As an example consider the timing-marked MSC shown in figure 8.3. The timing-marked MSC specifies a timing constraint between the notification time of the *LightSensor* component and the *Log* component. The corresponding DFA is depicted in figure 8.4.

Tracking the progress of a constraint We have seen in chapter 3 that when a message with a hidden parameter is processed, the hidden parameter is retransmitted with every message that is send. We will now slightly modify the matching method explained in MBT:

1. if the message that is matched carries a unique identifier then
   (a) lookup the DFA of the timing-marked MSC.
   (b) if there is a match between the message and the DFA

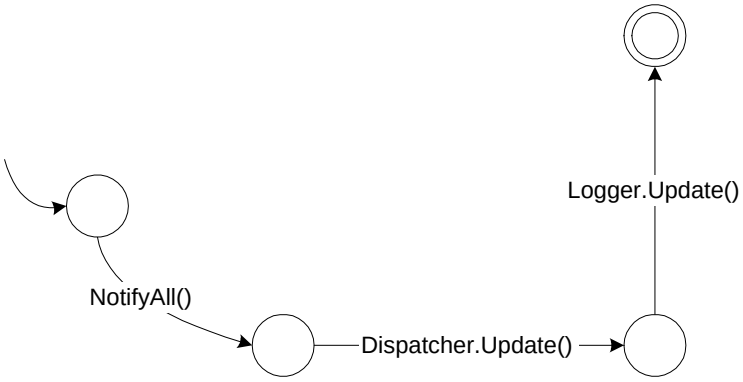

Figure 8.4: DFA of the Constraint

i. then move the following pointer to the next state
ii. else remove the hidden parameter

Figure 8.5 shows the cooperation between the MBT and CBT. When the MBT detects a hidden parameter carrying a unique identifier the CBT is informed and can update the constraint DFA it is tracking. It is obvious that we can use the same optimisation techniques to match the transitions of the timing-marked MSC as explained in the previous section. Another optimisation that we have made here is to remove the hidden parameter. Since the message didn't match with the timing constraint it will not match in the future either, so when it is removed we have no extra overhead.

8.3 Requirements for Real-Time Software

With each message that is sent by a component some run-time overhead is created. We want to minimize the overhead in each program that has such an amount of overhead at run-time. The overhead can be minimized by using POOL-structures and lookup tables as explained above. A more important issue is that the overhead created from tracking should have a fixed upperbound. This is an issue because the overhead should be taken into account when doing a schedulability analysis of the program. The overhead can be taken into account in the schedulability analysis by adding the overhead of sending a message to the execution time of that message.

8.4 Summary

If we want to generate a scheduler that can make decisions based on the state of the system we can build a tracker. The tracker used in this dissertation will follow the execution state of the different components by monitoring

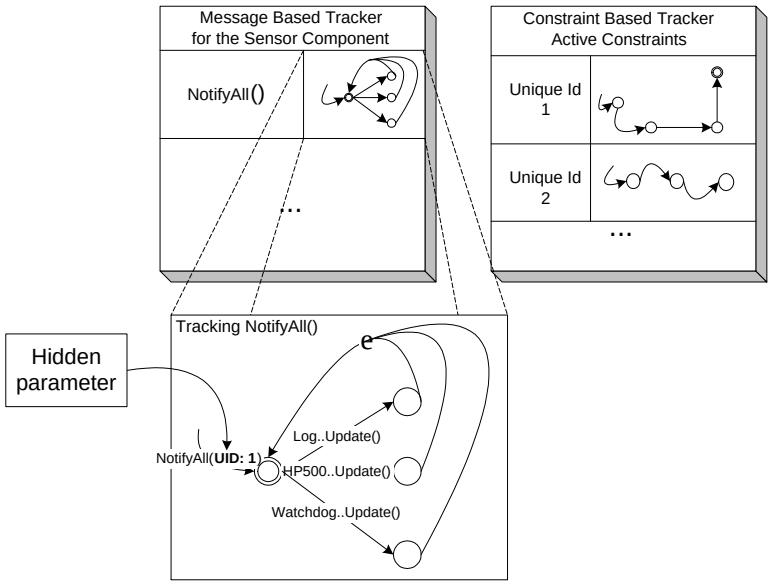

Figure 8.5: Relation between MBT and CBT

the messages that are send between the components. The tracker system works on two levels, namely message-based tracking and constraint based tracking. The message-based tracker uses concrete MSCs that are converted into a DFA. The tracker can then provide information about the state of the system to the scheduler. The role of the tracker in the component system is depicted in figure 1.4 on page 10. A transition in the DFA is done by matching the message that was send. The matching of the messages can be optimised to an O(1) operation by using lookup-tables. The constraint-based tracker uses the states of the DFAs extracted from the concrete MSCs to follow the constraints defined in the timing-marked MSCs.

Chapter 9

Real-Time Scheduling

In this chapter we will discuss scheduling on a single-processor system. Scheduling is the process of choosing which concurrent process becomes active. A scheduler is used in a software system for enforcing different timing strategies. This can vary from enforcing a timing constraint to implementing a fairness strategy among different concurrent parts. We discuss two ways how we can do scheduling in the component system. One possibility is to use priority-driven scheduling algorithms. Another possibility to do scheduling is to let the component choose what message it will process next. The decisions on how we can assign different thread priorities and what message should be processed by a component are taken by using the information provided by the tracker discussed in the previous chapter. First we discuss the information that is available to make some scheduling decisions.

9.1 Available Information

Scheduling decisions are based upon some information that is available or that can be predicted. The information should provide the scheduler with knowledge about the future. A scheduler is called clearvoyant if it knows everything about the future.

The tracker that is described in the previous chapter provides us with some information that can be used to make the scheduler a bit more clearvoyant. Below is a summary about what information is available. By using the tracker we know:

- when a timing constraint started (e.g. when a component sent a message that activated a timing constraint specified in a timing-marked MSC)

- how much time is left before the deadline of a timing constraint expires (e.g. when a timing constraint is detected, the starting time can be

saved. At each point in time we can calculate the time between the deadline and the current time)

- the components on which the system relies in order to complete a task (e.g. we know when a component is waiting for another component, before it can continue with its execution)

- sometimes we can predict that a timing constraint will occur (e.g. when a path unambiguously leads to the start of a timing constraint, by interpreting the periods of the time-triggered messages)

- the periods of the time-triggered messages (e.g. the period that is denoted in the abstract MSCs)

- the set of messages that could be sent in the future (e.g. the messages that are denoted in the concrete MSCs of the message)

9.2 Priority-Driven Scheduling Algorithms

A priority-driven scheduling algorithm does scheduling by chosing priorities for different concurrent tasks, also called threads. Whenever a thread gets a higher priority than the one thread that is running, the running thread is immediately interrupted (also called preempted) and the thread with the higher priority can run now. There exist several strategies in which we can update the priorities of the different tasks.

Definition 21 (Optimal Scheduler) *An Optimal Scheduling Algorithm finds a schedule if the best clearvoyant scheduler can find a schedule.*

Definition 22 (Feasibility) *A Scheduling Algorithm is called feasible if it can find a schedule so that no deadline is missed.*

We can make the distinction between three different scheduling algorithms:

1. Static scheduling algorithms

2. Dynamic scheduling algorithms

3. Hybrid scheduling algorithms

We discuss these three classes of algorithms and some of the research [21, 23, 24] that is available about them.

9.2.1 Static Scheduling Algorithms

In a static scheduling algorithm, all threads are assigned a fixed priority at compile-time. Such algorithms can only be used in software that has a fixed temporal task structure. This causes the algorithm to be inflexible to changing situations in the environment. An advantage however, is that a static scheduling algorithm does not create any run-time overhead. A static scheduling algorithm can also be seen as a sufficient schedulability test. The static scheduling algorithm called Rate Monotonic Scheduling is discussed in [21], the results are summarised below.

Rate Monotonic

In Rate Monotonic Scheduling (RM) the priorities are chosen at compile-time using the periodicities of the tasks. The requests for all tasks must be periodic[1], with a constant interval between requests and the periodicities are denoted with T1, T2, ..., Tm. The execution times of the different tasks are fixed and are denoted with C1, C2, ..., Cm. (m is the number of concurrent tasks). The task with the shortest period gets the highest priority. The rate monotonic scheduling is optimal, which means that it will find a feasible schedule if one exists. A sufficient condition for schedulability is given by:

$$\sum_{i=1}^{m} \frac{Ci}{Ti} < m(2^{\frac{1}{m}} - 1)$$

For large values of m:

$$\sum_{i=1}^{m} \frac{Ci}{Ti} < ln2 = 0.693$$

The last result shows that the least upper-bound for the processor utilisation is about 70 percent. This means that about 30 percent of the processor power is not utilised, which is a disadvantage when we look from an economical perspective. It is however possible in many situations to increase the processor utilisation to 90 percent. This can be accomplished by using a buffer mechanism [21] for a number of tasks with a lower priority and by relaxing their deadlines.

9.2.2 Dynamic Scheduling Algorithms

In a dynamic scheduler all scheduling decisions are computed at run-time. This implies that we have some overhead in choosing and altering the different thread priorities at run-time. Below we discuss some dynamic scheduling algorithms.

[1] Sporadic tasks can also be used by considering them as a Periodic tasks [19]

Earliest Deadline First

In [21] a dynamic scheduling algorithm is also studied for use in hard real-time systems. In this algorithm the priorities are assigned according to the deadlines of the current request. A thread is assigned the highest priority if the deadline of its request is the nearest or the thread will be assigned the lowest priority if the deadline of the request is the furthest. This means that at any time a thread with the nearest deadline, that is not completed yet, will get the access to the CPU. The priorities of the thread can change over time and the algorithm is thus called dynamic. The paper gives a proof that this algorithm is optimal, just like the Rate Monotonic Scheduling algorithm. In [21] a proof is also given of a necessary and sufficient condition for the feasibility of the algorithm:

If C1, C2, ..., Cm are the worst case execution times of the m concurrent tasks and T1, T2, ..., Tm are the periods of these tasks then the Earliest Deadline First algorithm is feasible if and only if:

$$\sum_{i=1}^{m} \frac{Ci}{Ti} \leq 1$$

The result also shows that the least upper bound of the processor utilisation is 100 percent. A serious disadvantage of the Earliest Deadline First algorithm is that it becomes unpredictable when the system is overloaded and deadlines are missed.

Least Laxity First

Least Laxity First is another dynamic scheduling algorithm. It is based on the laxity of the different deadlines.

Definition 23 (Laxity of a Deadline) *The Laxity of a Deadline for a task T is defined by:*

$$Laxity(T) = Deadline(T) - CurrentTime - ExecutionTime(T).$$

Intuitively the laxity of a task is the time that we can waste without failing to meet the deadline.

The task with the least laxity gets the highest priority. The accuracy of the algorithm depends on the prediction of the execution time. When the prediction of the execution time is estimated higher than the real execution time a task can get wrongfully a higher priority. The Least Laxity First algorithm becomes unstable in an overloaded system, just like the Earliest Deadline First algorithm.

9.2.3 Hybrid Scheduling Algorithms

In a hybrid scheduling algorithm the process of chosing priorities for different threads is done by mixing a static scheduling algorithm with a dynamic scheduling algorithm.

Maximum Urgency First

The scheduling algorithms discussed above have the disadvantage that they become unstable in overloaded situations. A hybrid scheduling algorithm that uses a combination of Rate Monotonic scheduling and Least Laxity First scheduling is proposed in [23]. The urgency of a task is defined as a combination of two fixed priorities and a dynamic priority. The two fixed priorities are called the *criticality* and the *user* priority. The process that becomes active is chosen with the following rules:

1. Select the task with highest criticality (fixed priority)
2. If two or more tasks share the highest criticality then the task with the highest dynamic priority (i.e. with the least laxity) is chosen. Tasks with no deadlines have a dynamic priority of zero.
3. If two or more tasks share the highest dynamic priority then the task with highest user priority is chosen.
4. If there is still no unique task, then the task is chosen on a first-come first-served basis.

We can choose the set of hard real-time constraints by assigning them the highest critical priority. The advantage of this system is that you can define a set of hard real-time constraints and a set of soft real-time constraints. This algorithm is considered stable, because a deadline miss will always occur in the set of soft real-time deadlines and will not affect the hard real-time deadlines.

9.3 Adjusting Thread Priorities using a DFA

The algorithms that are explained above cannot be applied "as is" to the component system we have defined. This is mainly because these algorithms are all based on the idea that one thread computes one specific (real-time) task. If we take a look at the architecture of our component system then you can see that when a component has to compute a specific task this involves delegation of subtasks to other components. As the communication is asynchronous between the different components, the task is actually computed by different threads, so modifying the priority of a single thread is not sufficient. When we assume that each component has one thread then

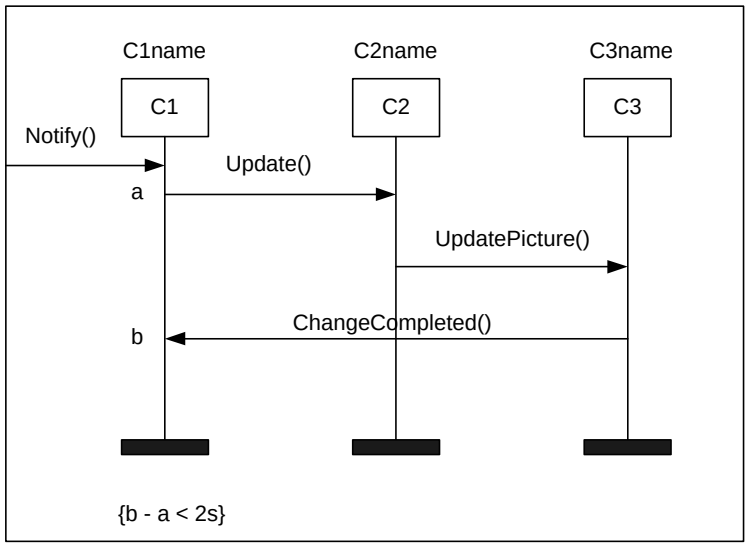

Figure 9.1: Example of a Timing-marked MSC

the task is actually computed by the different threads. It is clear that we need to identify all the threads that are involved in the computation of that single task.

9.3.1 Time-Constrained Components

When we consider the timing-marked MSC shown in figure 9.1 then we can determine the set of components the system relies on in order to satisfy the timing constraint. The set of components is C1name, C2name and C3name. Each time a message is sent, this set of components could change. This is illustrated in table 9.1. The dependent component for a timing-marked MSC can be computed at compile-time. The idea is to store them in the DFA that is extracted from the timing-marked MSC shown in figure 9.2. At each point in time we know for each timing constraint what components will have to process requests. With that kind of information we can change the priorities of the threads that serve those components. With each change in the running software we can update the different priorities easily.

9.3.2 Adjusting the priorities

By using a DFA as described above we know at each point in time the components that are used to satisfy a certain time constraint. The different scheduling algorithms that were discussed above could be used on the set of dependent components. When a certain time-constrained task is requested or initiated by a triggered message then the tracker can detect this and

	Message Send	Dependent Components
1	Notify()	C1name, C2name, C3name
2	Update()	C1name, C2name, C3name
3	UpdatePicture()	C1name, C3name
4	ChangeCompleted()	C1name

Table 9.1: Dependent Components

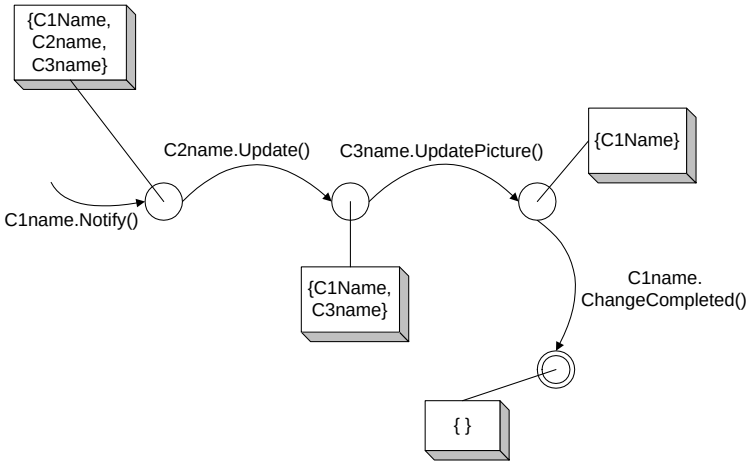

Figure 9.2: DFA with the dependencies

apply one of the scheduling algorithms on the set of dependent components. As the computation of the task evolves we have to change the priorities of the different threads. To satisfy the timing-constraint depicted in the timing-marked MSC shown in figure 9.1 we have to alter the priorities of the component's thread in the following order:

	Message Send	Components Thread
1	Notify()	C1name
2	Update()	C2name
3	UpdatePicture()	C3name
4	ChangeCompleted()	C1name

9.3.3 Scheduling using the Real-Time Behaviour

Besides the time constraints we introduced the description of different other constructs that influence the timing behaviour of a system in chapter 6. We discuss the problems they cause and how we solve them:

- Dependencies
- Time-Triggered Messages
- Event-Triggered Messages

Dependencies

Dependencies cause a component to wait for a message from another component before it can continue with its execution. They can cause a priority inversion, which means that a task with a higher priority is waiting for a task with a lower priority. Reconsider the timing-marked MSC from figure 9.1 and the concrete MSC shown in figure 9.3. Figure 9.3 shows that component C2 will wait for the C2name..Update() message *and* the C2name..UpdateLayout() message before it will send the UpdatePicture() message. When the timing constraint shown in figure 9.1 occurs then it will wait for the component C4name to send an UpdateLayout() message. Now assume that C4name runs at a lower priority than the timing constrained task.

We can prevent a priority inversion from happening by doing the following:

1. Identify the set of timingmarked MSCs that have dependent message.
2. For each dependent message compute all execution traces that cause the dependent message to be send
3. Track the set of computed execution traces

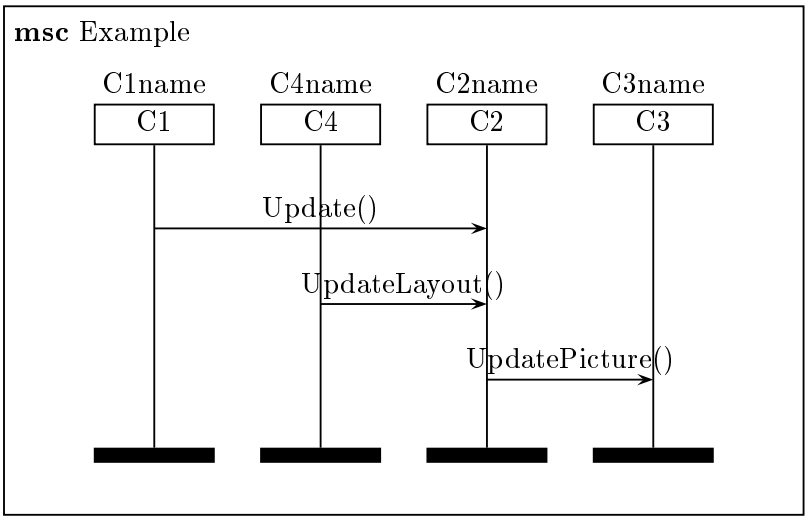

Figure 9.3: Example: Dependent UpdatePicture() message

4. Speed-up the computed execution traces when they are detected (by applying priority-inheritence)

Steps (1) and (2) can be computed at compile-time. The resulting set of computed execution traces can be tracked using the CBT algorithm described in the previous chapter. When a component is waiting for a certain dependent message the tracker can detect this and "speed up" the execution traces that cause the message to be send. The speeding up of the execution traces can be done by using priority inheritance [24] principles. Priority inheritance is the process of assigning the priority of the waiting component to the execution trace it is waiting for.

Triggered Messages and Loops

In section 6.1 we have seen two different message triggers:

1. Time-Triggered Messages
2. Event-Triggered Messages

When a message is sent it is either processed immediately by the component or it is put into a queue. If a certain message is recurrent and the component cannot process the message before the message recurs then the queue will not be bounded anymore. This implies that the responsiveness of that component cannot be guaranteed anymore, which is unacceptable when the component is used in a timing constraint. The same thing happens when a loop is entered where different messages are send to a component. We

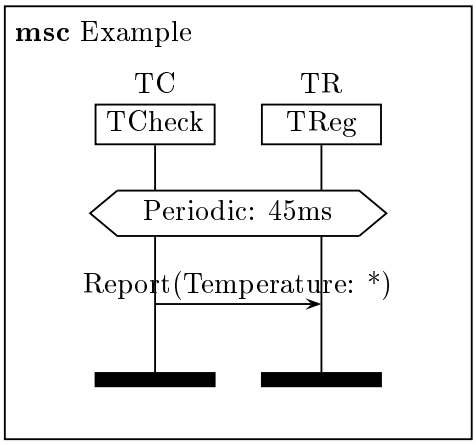

Figure 9.4: Example: Controlsystem of Nuclear Power Plant

must prevent that the queues from Time-Constrained components build up over time. This can be done by putting a monitor on these queues. When a queue exceeds its treshold the priority can be increased. The treshold is proportional to the laxity of the constraints that are to be served by the component, that is the lower the laxity of the constraint the lower the treshold of that queue. If no deadline misses are allowed the treshold must be set to the least laxity of all the constraints served on that component.

9.3.4 The Role of Refined MSCs

In section 4.5 we have seen how we can refine MSCs by taking the parameters into account. We will now discuss the benefits of refined MSCs used in the timing-marked MSCs. Sometimes a certain time constraint applies only when some certain preconditions are met. Consider a control system of a nuclear power plant. When the temperature rises above a certain treshold a series of actions need to be taken to cool down the system fast. When the temperature is below that treshold the temperature is written to a log file by the same component. It is obvious that the former case is urgent while the latter case does not have priority over the other processes. The concrete MSC of this case is depicted in figure 9.4.

With a refined timing-marked MSC we can specify that the timing constraint is only valid when the temperature is above the treshold. The refined MSC has the advantage that the system will only modify the different priorities in the case it is needed. Without this refinement, the system will not miss any deadlines but it will work harder than it should. This could cause other deadlines to be missed while they could have been met.

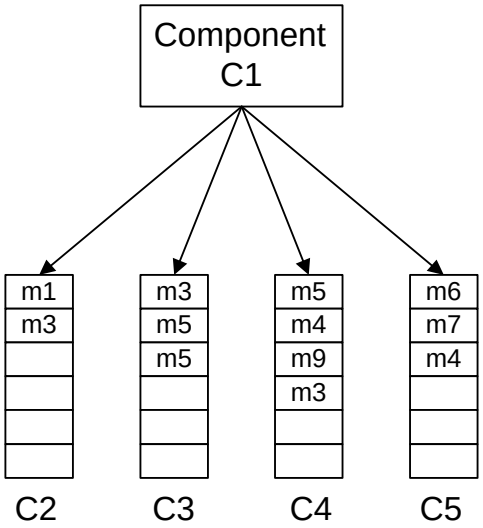

Figure 9.5: Internal Queue Structure

9.4 Scheduling with Queues

In chapter 3 we have seen that the order in which the messages are processed by a component needs to be the order in which they are sent by the different components. This constraint was necessary to preserve a certain logic in the temporal order of the messages that are sent. To maintain this restriction we can multiplex the queue of the component in different queues. After multiplexing the component queue there is one queue per client component (see figure 9.5). Suppose the component C1 has components C2, C3, C4 and C5 as clients. The messages sent to C1 by C2 are placed in the queue labelled C2, the messages sent by C3 are placed in the queue labelled C3 and so forth for each client component. Each time the component C1 has to choose the message it will process it has to select the queue from which it will pop the next message. It is clear that this choice affects the timing behaviour of the different time constraints. I propose to add priority numbers to the different queues, similar to the way threads have a priority number. When a component needs to select a message it will process the message from the queue with the highest priority.

The priorities of the different queues can be assigned in a similar way priorities are assigned to the different threads. For the assignment of the priorities one of the scheduling algorithms could be used in combination with the tracker.

9.5 Summary

Thread-based software can be scheduled by using a priority-driven scheduling algorithm. A priority-driven scheduling algorithm changes the priorities of the threads. The CPU is divided over the threads with the highest priority. There are three classes of priority-driven scheduling algorithms: static, dynamic and hybrid algorithms. These algorithms cannot be applied "as is" to the component system, because of its asynchronous communication. By using information provided by the tracker and the component documentation (the concrete MSCs and the timing-marked MSCs) we can decide which thread has to be changed. This is shown in figure 1.4 on page 10. The changes to the priorities can be applied according to one of the scheduling algorithms. The tracker can also detect cases of priority inversion. A priority inversion occurs when a component with a higher priority is waiting for a component with a lower priority. When a priority inversion is detected by the tracker it can be resolved by giving the dependent components a higher priority. A second possibility to schedule within the component system is by multiplexing the queue of a component into seperate queues according to the sender of the message. Each multiplexed queue gets a priority and the component choses the queue with the highest priority. The queues can be assigned a priority using one of the priority-driven scheduling algorithms and the tracker.

Chapter 10

Example: A Real-Time Simulation

To test the applicability of the methods described above we have implemented a testcase in the form of a Real-Time simulation. Simulations is an area that has many important applications (e.g. automotive industry, space travelling, astronomy, engineering, ...). A simulation can for example be used to test some properties of new building constructs before they are actually build. This way they can detect flaws in the design without actually building them. The simulation we have implemented is based on the conduit system used in [18]. We have added some real-time properties to this simulation. In this chapter we will work out the different aspects of the simulation.

10.1 The Basic System

In the conduit system you have a number of limited building blocks, namely a pipe, a join and a pump. These are the three components used for building a conduit system. Each component can be configured to have some specific properties. A pipe has for example a length, a capacity, ... The simulation is hooked up to a graphical component in the form of a HTTP daemon. So it is possible to follow the progress of the simulation by browsing to some webpage. The HTTP daemon is also a component, in which different add-on CGI-components can be plugged. An UML component diagram is shown in figure 10.1. The configuration of the conduit system used in the example is shown in figure 10.2.

10.2 Component Properties

I will now explain how the different conduit components interact with each other. Each conduit component accepts three messages related to the sim-

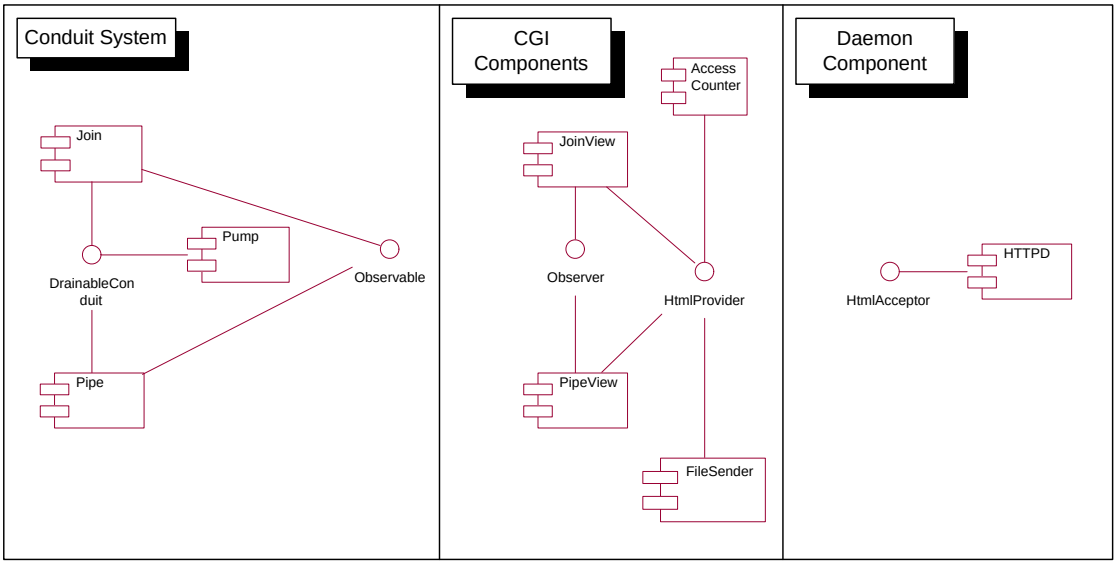

Figure 10.1: UML Component Diagram of the System

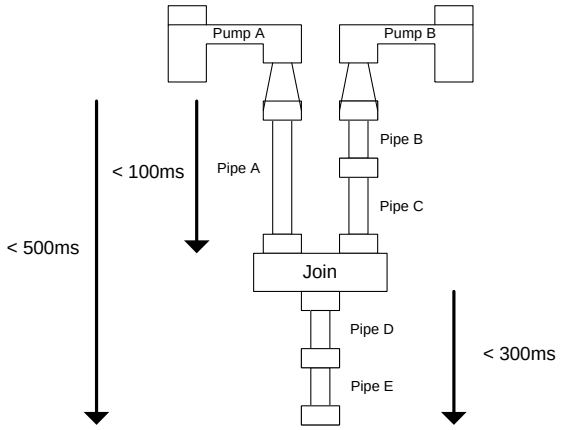

Figure 10.2: Configuration of the Conduit System

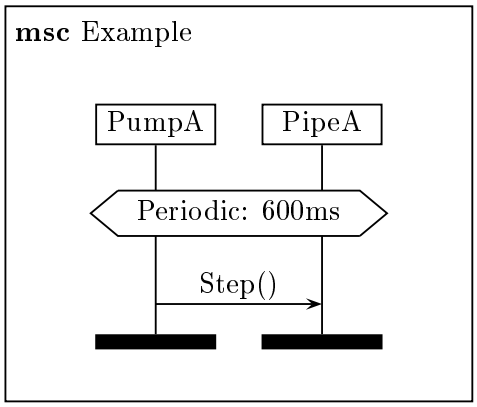

Figure 10.3: Periodic Step Message for Pump1

ulation:

1. *Step*: is sent with each tick of the clock and informs the component that it can update itself.

2. *Propose*: is sent to a conduit to propose a certain amount of liquid

3. *Take*: this message is a reply to a *propose* message to inform how much liquid the conduit will/can actually take.

The abstract communication behaviour of the components used in the experiment is shown in appendix B. We will now have a closer look at the more specific properties of the different components involved in the experiment.

10.2.1 Pump Component

A pump component has no input conduit and introduces liquid into the conduit system. The liquid is introduced into the system at well-defined time-intervals. The component also has a throughput property that indicates the maximum amount of liquid that can be passed at each time interval. In the simulation two pump components are used and they are both set at a time interval of 600ms. The time intervals are expressed with a time-triggered message as shown in figure 10.3

10.2.2 Pipe Component

The properties of a pipe are its length and the throughput capacity. Each pipe is divided in compartments. When some liquid is dropped in the pipe it goes from one compartment into the other until the end of the pipe is reached. When the liquid is at the bottom of the pipe it is *proposed* to

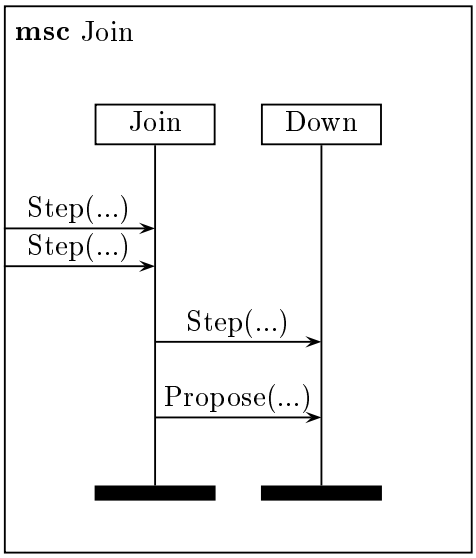

Figure 10.4: Join: Dependencies

the next conduit in the simulation. The abstract MSCs can be found in appendix B.

10.2.3 Join Component

A join component takes two input conduits and joins both inputs to one output. Such components are interesting to consider, because they can only output the liquid after they are served by both input pipes. This behaviour is comparable to the dependencies we have described in section 6.3. The dependency behaviour for the step function is expressed in figure 10.4.

10.2.4 Graphical Components

A graphical component can also be plugged into the system for some of the conduit components. These components provide a graphical representation of that specific component. The conduit component and the graphical components interact according to the observer-pattern described in [16]. In the testcase the graphical components actually produce an image encoded with JPEG-compression and adheres to the HTML-Provider interface as shown in figure 10.1.

10.3 Real-Time Parts

Typically in real-time software only a few tasks should run within some time-constraints while the other parts of the system can run without time

boundaries. In the experiment we will try to run the *simulation* in real-time, while the graphical user interface does not have to keep up with the time boundaries of the simulation. The graphical components will display correct information, but it could be that the information is delayed compared to the conduit components. The Httpd component and other CGI-components do not necessarily have to be considered by the scheduler, because they do not interact with the conduit components.

To make the simulation real-time we will try to enforce the three different timing constraints:

1. The program has to update the conduit components *PipeA, JoinA, PipeD and PipeE* at most 500ms after liquid is introduced into the conduits by *PumpA*.

2. The program has to update the conduit component *PipeA* at most 100ms after liquid is introduced into the conduits by *PumpA*.

3. The program has to update the conduit components *PipeD and PipeE* at most 300ms after liquid is introduced in *PipeD* by the *JoinA*.

The constraints are graphically expressed in figure 10.2. The next step is to define the time constraints using the MSCs. We know that the transfer between two conduits is realised by sending a *Propose* message that is replied to with a *Take* message. This process of negotiating how much liquid a conduit will take is initiated with the *Step* message. Consider the timing-marked MSC shown in figure 10.5. It reads as:

> The time between the event that PumpA sends a Step() message to PipeA and PipeE sends a take message to PipeD must be lower than 500ms.

This constraint expresses that the PipeE component must be updated at most 500ms after PumpA produced the liquid, so it only covers part of the constraint that we defined. The additional timing-marked MSCs that are needed to express the first timing constraint can be found in appendix C. The other timing constraints can be expressed similarly.

10.4 Handling the Dependencies

The first timing constraint from the previous section is dependent on the components *PumpB, PipeB and PipeC*. This is caused by the *JoinA* component that will wait for liquid from *PipeC* before it can propose liquid to the *PipeD* component. The *Step()* message of the *JoinA* is thus dependent on the *Step()* messages from the components *PipeA and PipeC*. This dependency is documented in the abstract MSC shown in figure 10.4. When the dependent message *Join..Step()* is detected in a timing-marked MSCs, the

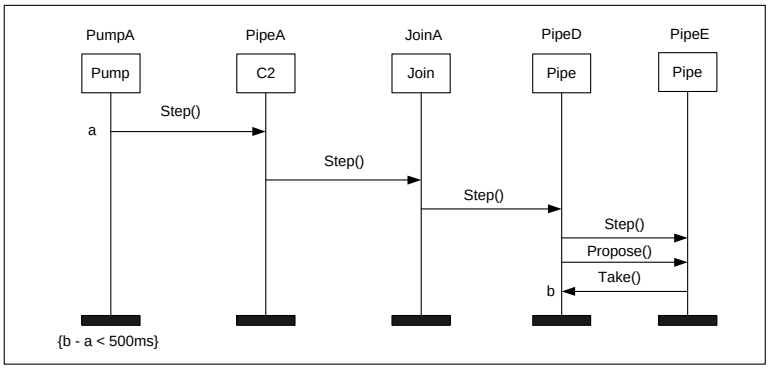

Figure 10.5: Timing-marked MSC for timing constraint

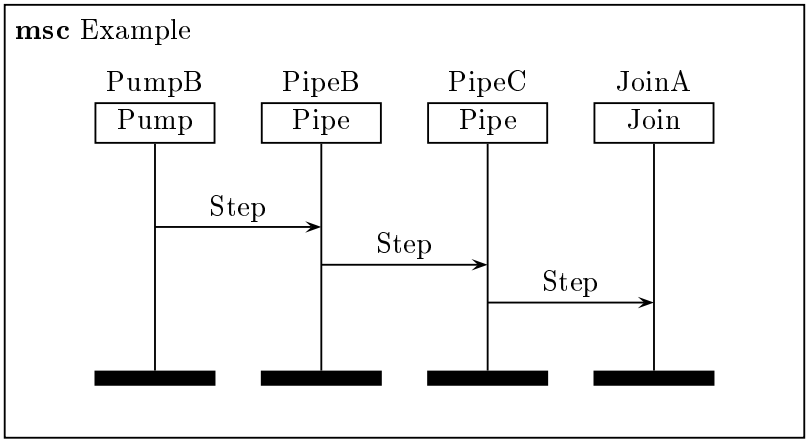

Figure 10.6: Dependent Path Computed for the Step() Message

path to activate the other *Step()* message is computed. This path is shown in figure 10.6. The path will be tracked by the CBT and when it is detected it will inherit the priority from the constraint that depends on it. When multiple constraints depend on that path it will inherit the highest priority of these constraints.

10.5 Summary

We have discussed the implementation of a real-time conduit simulation using the techniques that were discussed in the previous chapters. We also discussed how the priority-inheritance mechanism works on this example. The experiments on the simulation were not finished at the time of writing. The experiments that still need to be done are described in section 12.3.

Chapter 11

Mapping m Components onto n Threads

In the previous chapters we have always assumed that each component has its own thread that allows the component to "consume" its incoming messages. When a software system is composed and we have a large number of components then we have a large number of threads in the system. To run different threads in a monoprocessor system the underlying operating system or virtual machine will emulate this concurrency. This emulation creates extra overhead on the system leaving less processor power for the software. In this chapter we will discuss how this thread emulation is done in most operating systems and virtual machines to show the amount of overhead that is introduced by the different threads. Finally we will show a technique to reduce the number of threads in the component system.

11.1 Context Switching

The technique used to emulate the different threads in a monoprocessor system is called context switching. To show the overhead that is introduced by switching between the different active threads we will explain what happens on low-level when we switch between two threads.

1. Determine when to switch threads:
   In a priority-driven scheduler threads are switched when a thread is assigned a priority that is higher than the thread currently executing. Threads that have the same highest priority are interleaved in a small time interval

2. Save the state of the current thread:
   We need to stop executing the current thread and save the state of that thread. The state of the thread is sometimes also called the *context* of the thread. The context of a thread is defined by:

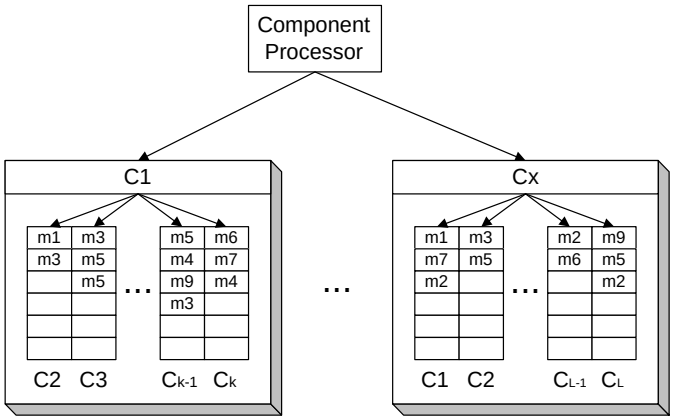

Figure 11.1: Message Processor Concept

- Program Counter
- Stack Pointer
- Other Registers

3. Choose the thread that needs to become active:
   In a priority-driven threading system, this is the thread with the highest priority

4. Restore the context of that thread

5. Resume executing the thread

These 5 steps are executed each time another thread becomes active. The overhead of switching between threads increases with the number of threads, because the number of switches increases with the number of threads.

11.2 Mapping Multiple Components onto 1 Thread

We can reduce the overhead of switching between the threads by mapping multiple components onto a single thread. We can extend the component system to support this by introducing the concept of a message processor. Each thread is a message processor that has to choose between the messages that were send to the different components that are mapped onto that thread. This concept is shown in figure 11.1.

11.3 Maintaining Schedulability

When different components are mapped onto a single thread we have to reconsider the scheduling algorithms. Suppose component X and component

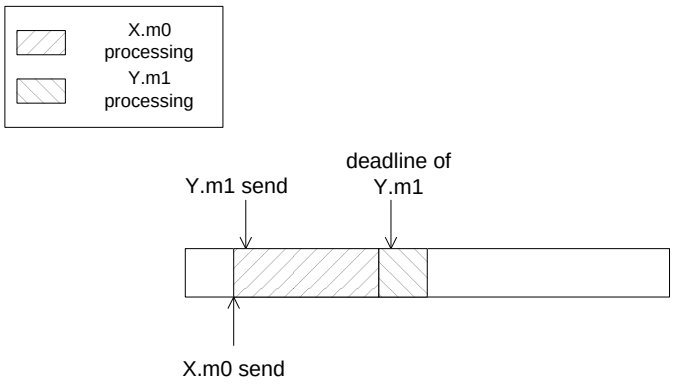

Figure 11.2: Example: Failing deadline on mapped components

Y are mapped onto the same thread. Message m0 of component X has a has a processing time of 300ms. Message m1 of component Y has a deadline of 300ms and a processing time of 200ms. Message m0 occurs a few milliseconds before m1. The message processor starts processing m0 and will fail to meet the deadline of message m1. This failure is mainly caused because the message processor cannot interrupt/preempt the processing of m0 when m1 arrives. This scenario is shown in figure 11.2. This example shows that the mapping of components that are involved in meeting deadlines is not an obvious task. To link the concept of mapping of components with the scheduling performance of the software we want to introduce the concept of an *equivalent mapping.*

Definition 24 (Equivalent Mapping) *A mapping of m components onto n threads with m >= n is called an* equivalent mapping *if a scheduling algorithm performs equally good or better as in a one-to-one mapping of the component onto threads.*

We have seen that one of the causes why we failed to meet this deadline was the fact that we had mapped a component that has to meet deadlines with another component.

Definition 25 (Real-Time Components) *Real-time components are involved in meeting deadlines. They are the components that are used in timing-marked MSCs and the components used in the dependent execution traces of these timing-marked MSCs.*

The set of real-time components can be computed from the different MSCs. We propose to give each component in this set its own thread. All the components that *do not* belong to this set can be mapped onto a single thread. We have no real proof that this is an equivalent mapping, but it

is easy to see that because each component that is involved in meeting a deadline has a seperate thread, it can therefore be preempted at any point in time.

When the number of threads still causes too many overhead in the system we have to find other mappings, but this is a beyond the scope of this thesis and is open for further research.

11.4 Summary

Having a one-to-one mapping of components onto threads can create too much overhead in a system that has many components. The number of context switches can be reduced by limiting the number of threads in that system. To limit the number of threads we have introduced the concept of a *Message Processor*. Each thread has a message processor. A message processor has to choose between the queues of the different components that are mapped onto his thread. The mapping of multiple components onto a thread is shown by the brace in figure 1.4 on page 10. The mapping of the components onto a thread is not an easy task. A wrong mapping can cause the system to miss its deadlines. An equivalent mapping should miss no more deadlines than a one-to-one mapping. By giving each component involved in the execution of real-time constraints a single thread and placing the other components onto one thread we might obtain an equivalent mapping. Finding equivalent mappings is open for further research.

Chapter 12

Conclusion

12.1 Summary

Embedded Systems A few years ago embedded software was not interesting from a research perspective, because they were simply too small. The last few years more research has been done in this area as they became larger and more difficult to develop, maintain and evolve. Existing software engineering methodologies for desktop applications cannot be applied to embedded software, because embedded software has special characteristics (such as robostness, temporal behaviour, ...). In this dissertation we considered the problems associated with reusability in soft real-time embedded systems that run on a single-processor system. In such systems the correctness of the software not only depends on the logical correctness of the program, but also on the time at which the results are produced. Real-time software is often concurrent by nature.

Adapted Components The latest paradigm that has reusability foremost in mind is called component-based development. Components are reusable entities that can be glued together to form a program. The problem is that components are not specifically designed for the creating embedded software. To match these characteristics an adapted component model has been proposed [10]. In this componentmodel every component should be thought of having its own thread of execution. All messages between the components are send asynchronously. By sending the messages asynchronous we can introduce concurrency implicitly into the system.

Component Documentation We propose to add the intercomponent communication as formal documentation to the components to solve the scheduling problems that are associated with embedded real-time software. Abstract components can be documented with abstract message sequence charts (MSCs). We had to extend the MSCs with new semantics, because

MSCs were intended for describing scenarios rather than the full communication behaviour. A concrete MSC describes the communication behaviour of concrete components. We can extract the concrete MSCs from the abstract MSCs when the components are glued together to form a system. When extracting the concrete MSCs from the abstract MSCs we try to resolve the variables that are present in the abstract MSC. A third class of MSCs are needed to specify the temporal behaviour of the components. The temporal behaviour can be specified with timing-marked MSCs. The MSCs are useful for doing temporal analysis and for making scheduling decisions.

Temporal Analysis Given the execution time of the component messages we can perform a schedulability analysis. With a schedulability analysis it is possible to test if the software composition can meet the deadlines that are specified in the timing-marked MSCs. Finding the execution times is a difficult task with many subtleties. Delivering the component with the execution time of its messages as documentation is not an option, because the components are likely to be reused on different hardware rendering the execution times useless. We propose to create a test set associated with each component. This way execution times can be measured without breaking the black-box principle of the components. When the components are running on a virtual machine we can also deliver the components with the number of processor ticks. The regular time can be computed given the number of processor ticks and the processor type. There are two classes of schedulability tests. A necessary and a sufficient schedulability test: when the former succeeds then the software composition will definitely meet its deadlines, while the latter will only fail when the software composition will definitely not meet its deadlines. Note that nothing can be said about the schedulability when the necessary test fails or the sufficient test succeeds. We have shown how we can perform a sufficient schedulability test given a timing-marked MSC and the execution times, which allows us to detect when deadlines will not be met by the current hardware configuration. This can be done by matching the execution times with the component communication that is necessary to complete the time-constrained task.

Tracking the Execution We can track the partial state of the software by monitoring the messages send between the components. The tracking of the system occurs on two levels: Message Based Tracking and Constraint Based Tracking. The former uses the concrete MSCs that were extracted from the abstract MSCs and the latter uses the timing-marked MSCs. The tracking of the software creates overhead which we can reduce by using lookup-tables and storing data at compile-time in the DFAs.

Scheduling We can schedule the component system by altering the priorities of the threads. With the tracker we can group the real-time tasks together and apply one of the existing scheduling algorithms. We can also detect dependencies, which can cause a priority inversion and resolve them by using priority inheritance. We can also schedule in the component system by multiplexing the queue of a component. The component will then choose to process its next message from the queue with the highest priority.

Mapping Multiple Component Onto a Single Thread When software is composed of many concrete components we have many threads. Overhead is created with each switch between the active threads. The amount of overhead can be limited by reducing the number of threads in the system. It is possible to reduce the number of threads in the component system by mapping multiple components onto a single thread. This can be implemented by using a message processor. A message processor has to choose between the queues of the components. The choice of which components are mapped onto a single thread can influence the temporal behaviour of the program and should be carefully considered.

12.2 Overall Conclusion

Asynchronous programming implicitly introduces concurrency into the system. By making concurrency implicit it becomes easier to reason about such systems. One drawback of asynchronous messages is that scheduling becomes harder, because computing a task is multiplexed over different threads in the system. When you look at the development of a thread-based system the software developers have to choose which parts will run in parallel. By putting a tracker in the system and documenting the components with semantic documentation we can make scheduling easier than in thread-based systems, without having the burden to make the system concurrent.

The writing of this dissertation has led to the following implementations:

- The first prototype of the component system. Later versions were developed by Werner Van Belle for the SEESCOA consortium
- A tool to extract abstract MSCs from component source code has been implemented by Werner Van Belle
- A parser for parsing abstract and concrete MSCs
- A tool for extracting the concrete MSCs from the abstract MSCs
- A Message Based Tracker and a Constraint Based Tracker have been implemented using the techniques laid out in chapter 8

- The component model has been extended to support measuring regular time with test sets
- A tool has been developed to perform a sufficient schedulability analysis as explained in chapter 7
- The Real-Time simulation from chapter 10 was implemented using most of the tools described above

The results of this dissertation exceeded our initial expectations. We have resolved many of the problems that are common to embedded software development and have presented a technique that allows components to be reused in real-time embedded software. This technique does not conflict with the black-box principles that are common to component-based development. Due to the limited amount of time however, some experiments have not been done. These experiments are described in the next section.

12.3 Future Work

This dissertation is a first step towards the reusability of software components in real-time embedded systems. The usage of a tracker has been explored in synchronous real-time systems and real-time distributed systems in [25, 26, 27], but no research was found about tracking asynchronous real-time systems. It is clear that the use of an asynchronous model has some interesting benefits. Nevertheless, a lot of research needs to be done, also experiments with a scheduler placed on top of the tracker (as explained in chapter 9).

12.3.1 Experiments

The experiment described in chapter 10 was not completed at the time of writing. More specifically the following experiments still need to be conducted:

- Test the performance of existing scheduling algorithms on top of the tracking system
- Test the influence on the performance of the component-to-thread mapping on the scheduling algorithms

The setup of the simulation described in chapter 10 could be used to conduct these experiments.

12.3.2 Future Research

In this dissertation we have touched the surface of several items that need further research:

- Finding an upperbound on the length of the queues
- The mapping of multiple components onto threads
- Scalability
- Software evolution

Finding an Upperbound on the Length of the Queues

When we want to use the component system in hard real-time systems we need to know the upperbound length for each of the queues. Finding an upperbound on the queue is necessary to guarantee a maximum response time for each message.

Mapping Multiple Components onto Threads

In chapter 11 we discussed that we can reduce the number of threads by introducing message processors into the component system. An equivalent mapping is a mapping of multiple components onto a smaller number of threads so that the temporal correctness of the system is not affected. We explained that a possible equivalent mapping could be constructed by mapping the components, involved in meeting deadlines, a separate thread. All the other components could then be mapped onto a single thread. This is one possible mapping, but when multiple deadlines are spread over the different components the number of threads are still too high.

Scalability

After expanding abstract MSCs to concrete MSCs, we could end up with large concrete MSCs. Large MSCs become difficult to read. Besides that the concrete MSCs are also stored as DFAs in the embedded software, so they could consume a lot of memory. It might be possible to reduce the amount of memory by simply not expanding all the concrete MSCs (e.g., we don't need the concrete MSC of a message that is not involved in the meeting the specified time constraints).

Software Evolution

The timing-marked MSCs need to be redefined each time a component, that is used in a timing constraint, is replaced in the software. The replaced component could use other asynchronous components to complete its tasks, so that the timing-marked MSC needs to be redefined by adding these components. This could be solved by adding another form of formal documentation that defines the tasks a component can perform and when these tasks are considered completed. The time constraints would then be

defined using the task-documentation rather than the timing-marked MSCs. The timing-marked MSCs could then be expanded from the tasks.

Appendix A

Conversion of Constructs to MSCs

The figures in this appendix show how the basic programming language constructs can be mapped in a abstract MSCs.

```
component SourceComponent
{
  message EventA()
  {
    if (<statement>)
    {
      DestinationComponent..EventB();
    }
  }
}
```

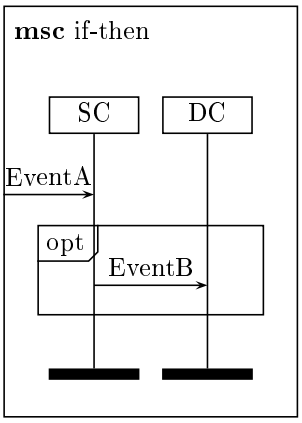

Figure A.1: Conversion of if-then constructs

```
component SourceComponent
{
  message EventA()
  {
    if (<statement>)
    {
      Component1..EventB();
    }
    else
    {
     Component2..EventC();
    }
  }
}
```

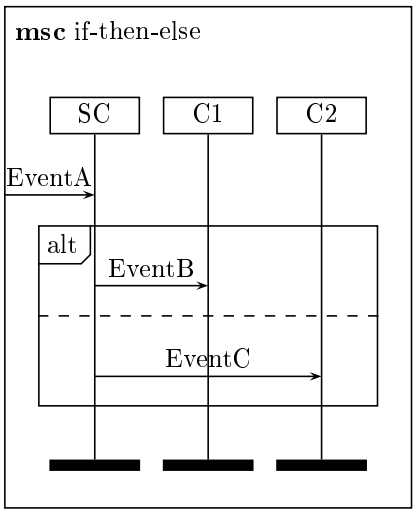

Figure A.2: Conversion of if-then-else constructs

```
component SourceComponent
{
  message EventA()
  {
    switch (<variable>)
    {
      case A:
        Component1..EventB();
        break;
      case B:
        Component2..EventC();
        break;
      .
      .
      .
      case Z:
        ComponentN..EventX();
        break;
    }
  }
}
```

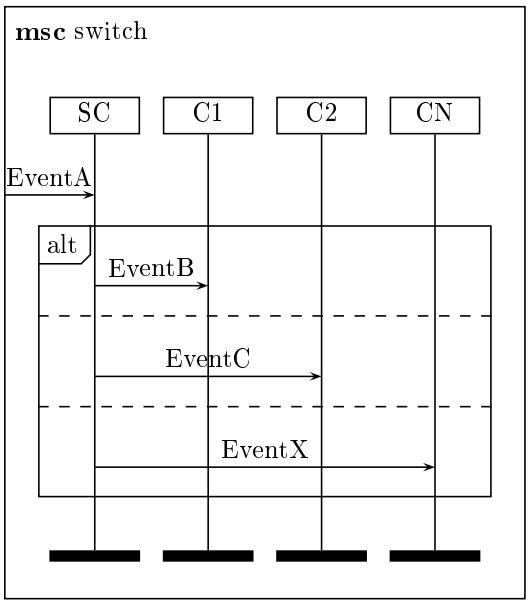

Figure A.3: Conversion of switch constructs

```
component SourceComponent
{
  message EventA()
  {
    while(<statement>)
    {
      Component1..EventB();
    }
  }
}
```

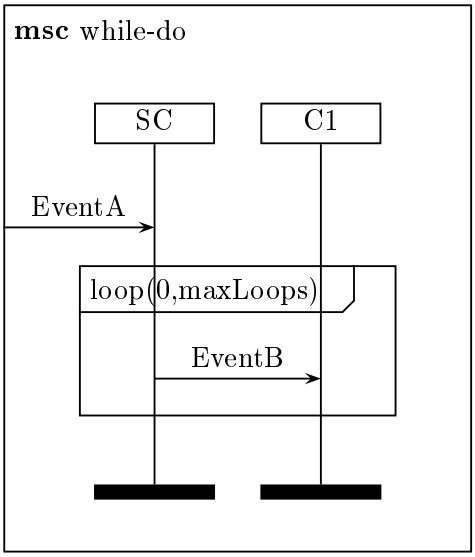

Figure A.4: Conversion of while-do constructs

```
component SourceComponent
{
  message EventA()
  {
    do
    {
      Component1..EventB();
    } while(<statement>);
  }
}
```

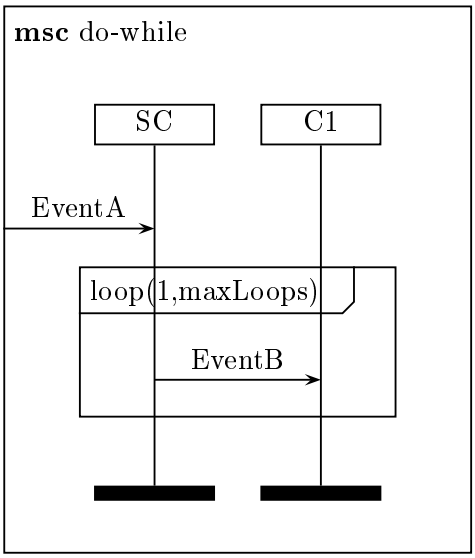

Figure A.5: Conversion of do-while constructs

Appendix B

Abstract MSCs of Conduit Components

This appendix contains the abstract MSCs that were extracted from the components used in the real-time simulation described in chapter 10.

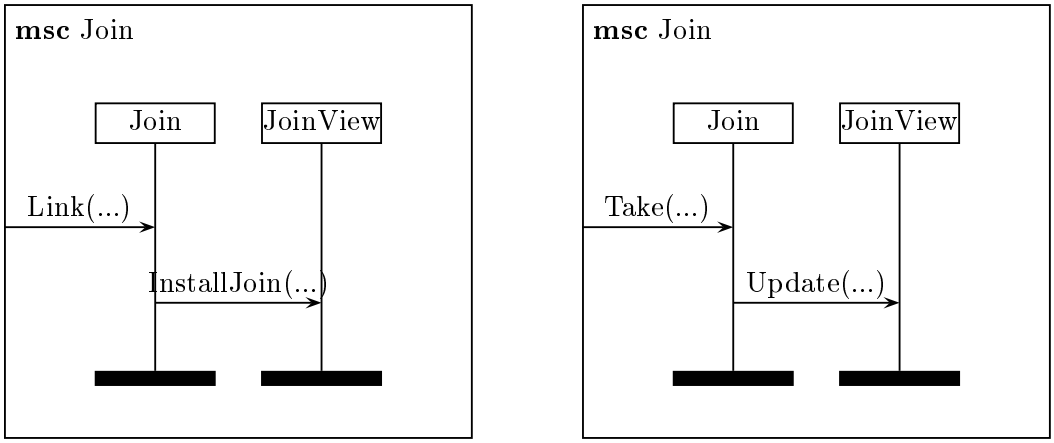

Figure B.1: Join: Link-Take

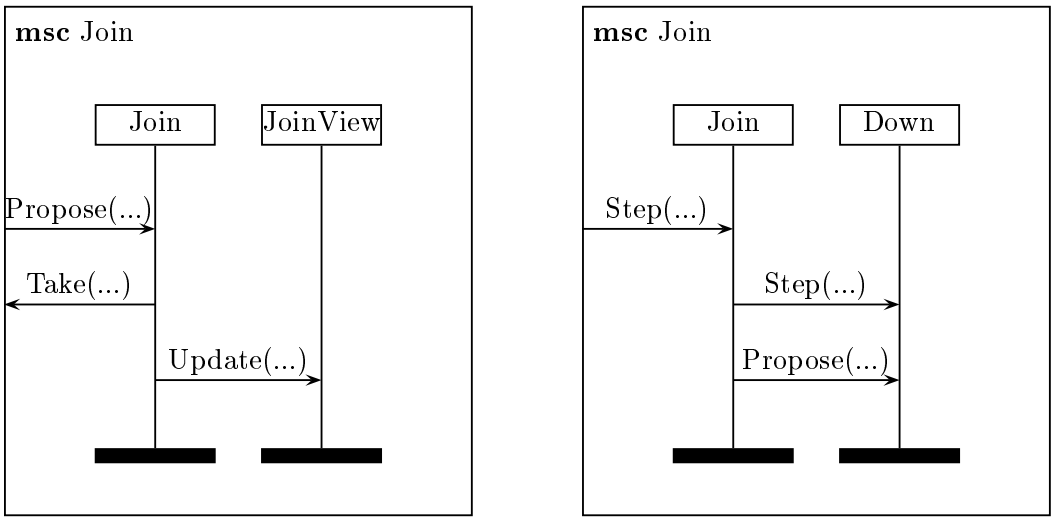

Figure B.2: Join: Propose-Step

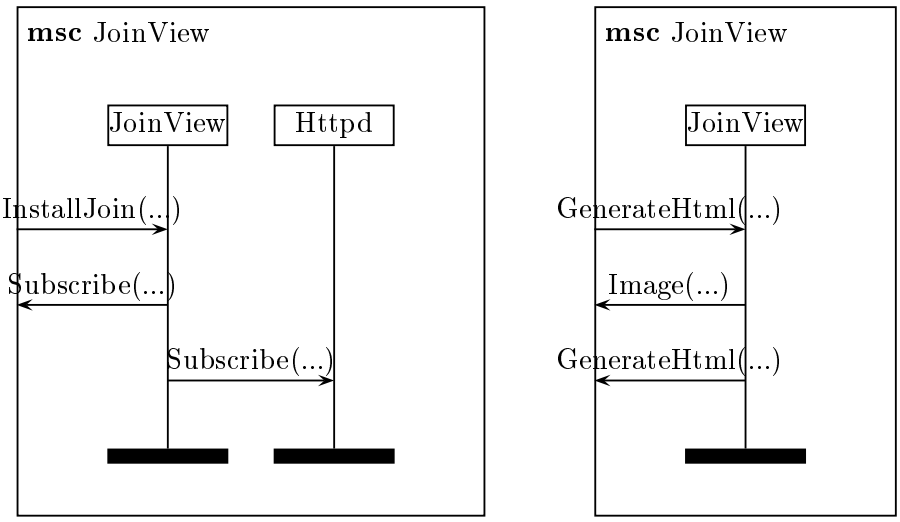

Figure B.3: JoinView: InstallJoin - GenerateHTML

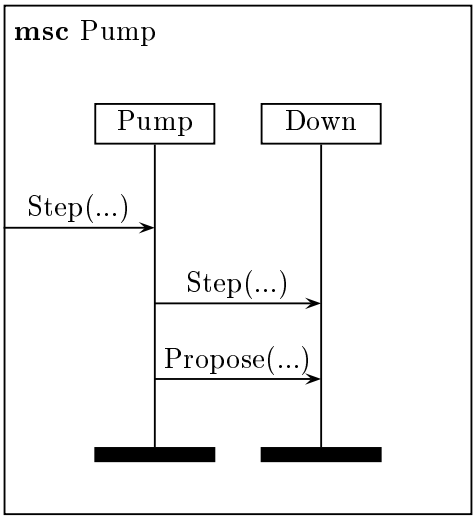

Figure B.4: Pump: Step

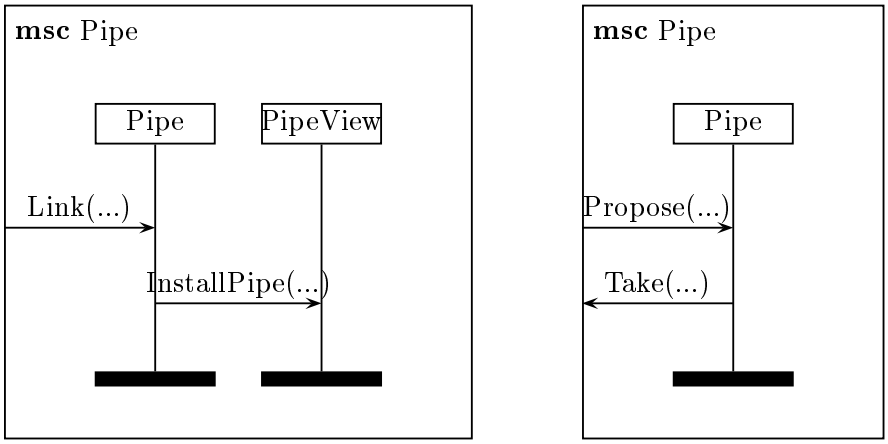

Figure B.5: Pipe: Link-Propose

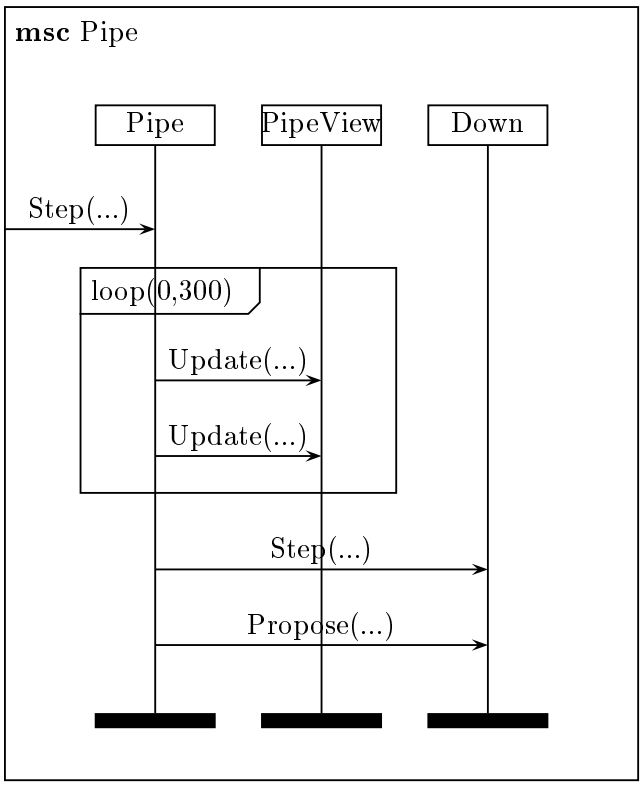

Figure B.6: Pipe: Step

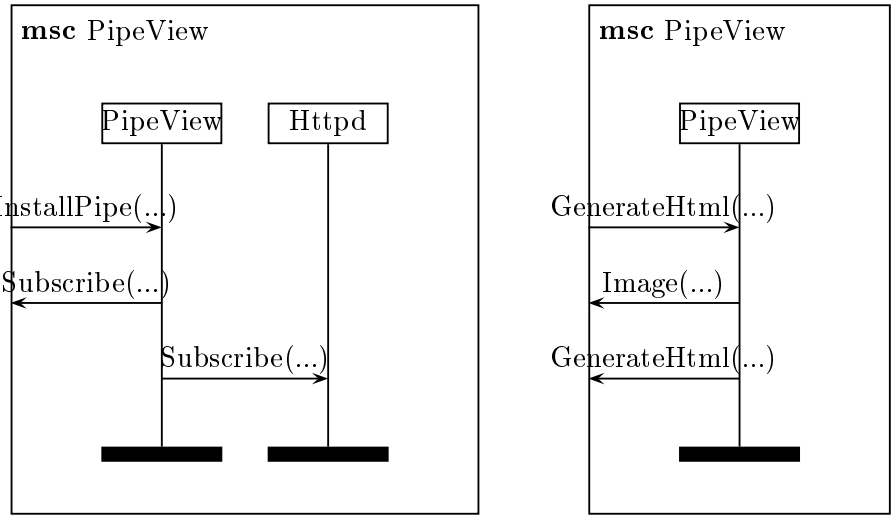

Figure B.7: PipeView: InstallPipe - GenerateHTML

Appendix C

Timing-Marked MSCs of ConduitSystem

The figures show the additional timing-marked MSCs that we needed to define the first time constraint that was used in the real-time simulation (chapter 10).

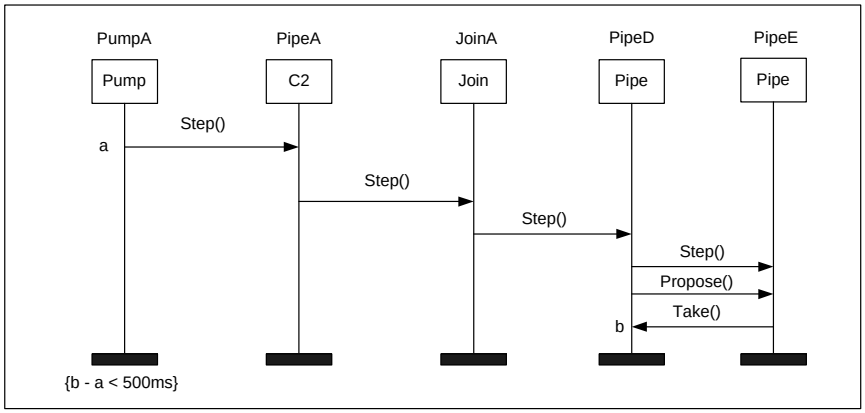

Figure C.1: Timing-marked MSC: PipeE needs to be updated within 500ms

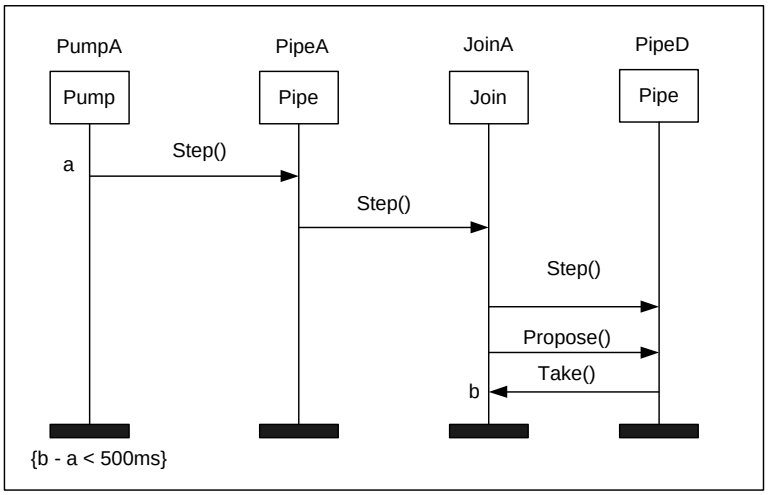

Figure C.2: Timing-marked MSC: PipeD needs to be updated within 500ms

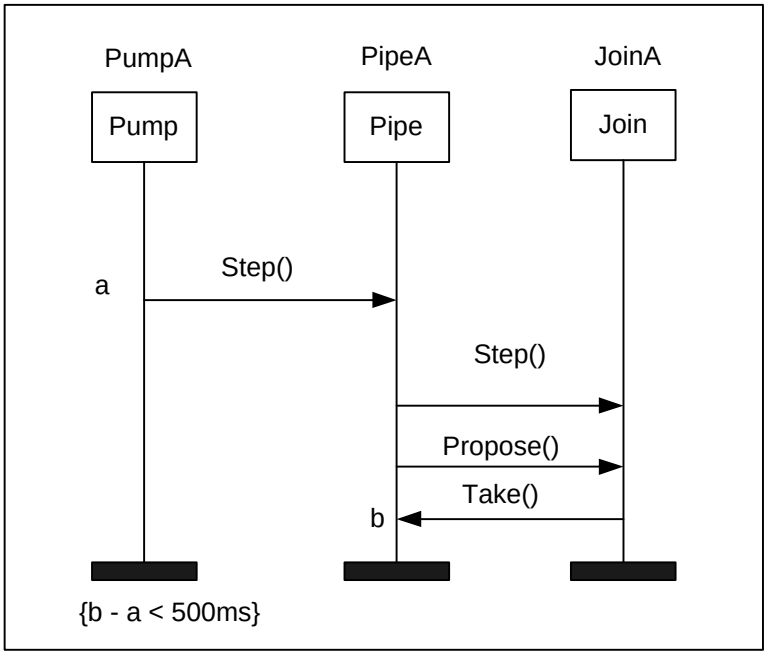

Figure C.3: Timing-marked MSC: JoinA needs to be updated within 500ms

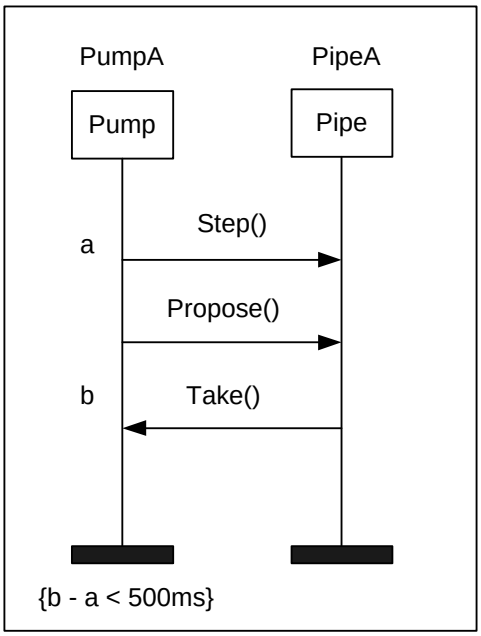

Figure C.4: Timing-marked MSC: PipeA needs to be updated within 500ms

Bibliography

[1] A. Birk, Autonomous Systems, Unpublished Draft, 2000

[2] SEESCOA: Software Engineering for Embedded Systems using a Component-Oriented Approach

[3] Deliverable D1.4 Working Definition of Components, SEESCOA - Software Engineering for Embedded Systems using a Component Oriented Approach.

[4] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," IEEE Computer, vol. 24, pp. 52–60, Aug. 1991.

[5] Agha, G. and Hewitt, C., Actors: Conceptual Foundation for Concurrent Object-Oriented Programming, in Wegner, P. and Shriver, B. eds. Research Directions in Object-Oriented Programming, MIT Press, Cambridge, MA, pp. 49-74, 1987

[6] Demaecker P., Combining Components using Control Flow Components, Programming Technology Lab, Vrije Universiteit Brussel, July 2000

[7] Maher Awad, Juuha Kuusela, Jurgen Ziegler, Object-Oriented Technology for Real-Time Systems, Prentice Hall, 1996

[8] Edward A. Lee, What's Ahead for Embedded Systems?, IEEE Computer Society

[9] Deliverable D2.1, Real-Time UML, SEESCOA - Software Engineering for Embedded Systems using a Component Oriented Approach.

[10] Deliverable D1.4, Working Definition of Components, SEESCOA - Software Engineering for Embedded Systems using a Component Oriented Approach.

[11] ITU-T. Recommendation Z.120, ITU - Telecommunication Standardization Sector, Geneva, Switzerland, May 1996.

[12] R. Alur, G. J. Holzmann, and D. Peled. An analyzer for message sequence charts. In T. Margaria and B. Steffen, editors, Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, Vol. 1055, pages 35–48. Springer Verlag, 1996.

[13] N. Meng-Siew. Reasoning with timing constraints in Message Sequence Charts. Master's thesis, University of Stirling, Scotland, U.K., August 1993.

[14] H. Ben-Abdallah, S. Leue. Expressing and Analyzing Timing Constraints in Message Sequence Chart Specifications. Technical Report 97-04, Department of electrical and Computer engineering, University of Waterloo, April 1997.

[15] Wydaeghe, B., Michiels, B. and Verschaeve, K. Documenting Components for Composition, November 1999

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns, Addison-Wesley, 1994

[17] Bruce Powel Douglass, Real-Time UML: Developing Efficient Objects for Embedded Systems, Addison-Wesley, 1999

[18] Principles of Object-Oriented Languages Course by Prof. Theo D'Hondt

[19] Real-Time Systems Course by Jacques Tyberghiens

[20] Dean Kelley, Automata and Formal Languages: An Introduction, Prentice-Hall, 1998

[21] C. L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a HardReal -Time Environment," Journal of the ACM, Vol. 20, No. 1, pp. 46-61, Jan. 1973.

[22] Hermann Streich: TaskPair-Scheduling: An Approach for Dynamic Real-Time Systems, Int. Journal of Mini & Microcomputers, Vol. 17, No. 2, pp 77-83, 1995.

[23] Stewart, D.B.; and Khosla, P.K. (May 1991). Real-Time Scheduling of Sensor-Based Control Systems. Proceedings of the IFAC/IFIP Workshop, May 15-17, pp. 139-144.

[24] Rajkumar, R. 1991. Synchronization In Real-Time Systems – A Priority Inheritance Approach. Kluwer Academic Publishers, Boston.

[25] Raju, S.C.V., R. Rajkumar, and F. Jahanian. Timing Constraints Monitoring in Distributed Real-Time Systems. in IEEE Real-Time Systems Symposium. 1992.

[26] Haban, D. and K.a.G. Shin, Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times. IEEE Transactions of Software Engineering, 1990. 16(12): p. 1374-1389.

[27] M. Gergeleit, M. Mock, E. Nett, J. Reumann: "Integrating Time-Aware CORBA Objects into Object-Oriented Real-Time Computations, Proc. of WORDS97, Newport Beach, USA, Feb.