Practical Verification of Real-Time Systems

ALEXANDRE DAVID

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA UNIVERSITY

Practical Verification of Real-Time Systems

BY
ALEXANDRE DAVID

September 2001

DEPARTMENT OF COMPUTER SYSTEMS
INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Science
at Uppsala University 2001

Practical Verification of Real-Time Systems

Alexandre David
adavid@docs.uu.se

Department of Computer Systems
Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

Formal methods are becoming mature enough to be used on non trivial examples. They are particularly well fitted for real-time systems whose correctness is defined in terms of correct responses at correct times. Most common real-time systems are of reasonable size and can therefore be handled by an automatic verification tool such as UPPAAL. Unfortunately the application of such techniques is not widely spread.

This thesis presents advances in making formal techniques more accessable technology for system development and analysis. As the first contribution, we report on an industrial case study to show that model checkers can be used for debugging and error localization. We shall present a number of abstraction techniques applied in the case study to avoid the state explosion problem. As the second contribution, we have developed a hierarchical extension of timed automata to enable more structured, compact, and more complex descriptions of systems by the users. Such a hierarchical representation is better suited for abstraction and expected to give better searching algorithms. Finally we present a hybrid animation system serving as a plug-in module for model-checkers to improve features for modelling and simulation.

List of Publications

- *Modeling and Analysis of a Commercial Field Bus Protocol*, Alexandre David and Wang Yi. In Proceedings of the 12th Euromicro Conference on Real Time Systems, Stockholm Sweden, 19-21 June 2000, pages 165-172. I participated in the discussions, made the models, conducted the verification and wrote the paper.
- *A Realtime Animator for Hybrid Systems*, Tobias Amnell, Alexandre David, Wang Yi. In proceedings of ACM SIGPLAN workshop LCTES , Vancouver June 2000, LNCS vol 1985. I participated in the discussions, implemented parts of the engine, and wrote the implementation section.
- *Formal Verification of UML Statecharts with Real-time Extensions*, Alexandre David, Oliver Möller, Wang Yi. Presented at the Nordic Workshop 2001, 10-12 October Denmark. Technical report, department of Information Technology, Uppsala University. I participated in the discussions and wrote the sections on syntax and semantics.
- *From* HUppaal *to* Uppaal*, A Translation from Hierarchical Timed automata to Flat Timed Automata*, Alexandre David and Oliver Möller. Technical report published in the BRICS report series, ISSN 0909-0878. I participated in the discussions and wrote the sections on syntax and semantics.

Acknowledgments

I would like to thank all the people who helped me during the work on this thesis. I cannot be complete because many have moved, but in an attempt they are: my supervisor Wang Yi for his patience, Paul Pettersson for his humour, the Uppsala part of the Uppaal team Elena Fersman, Johan Bengtson, Tobias Amnell, Fredrik Larsson, and Leonid Mokrushin for their support, the Aalborg part of Uppaal team Kim Larsen, Gerd Behrman, Oliver Möller, and Kåre Kristoffersen for their support.

I am very grateful to Ulf Hammar and Thomas Lindström for the time they spent in discussing implementation issues. I would like also to thank Julien d'Orso, Johann Deneux, and Sergei Vorobyov for their help and support, Helena Pettersson and Anne Marie Nilson who know all about administration and forms, and saved me many times. I thank my parents and my brother for giving me courage. Finally I apologize for those not being named here. I would probably need a book for you all.

The work has been supported by ARTES and ASTEC.

Introduction

This thesis consists of 3 parts: a case study on the analysis of a field protocol that is commercial product, the work on hierarchical timed automata done in collaboration with Aalborg University, and the work on hybrid automata.

The problem we are interested in is to analyse real-time systems. Due to the size and complexity of real systems, it is difficult to model and verify these systems. This work aims at presenting a technique for modeling large systems through a case study. This technique decomposes the system in different parts that are abstracted and put together. To improve the modeling and the verification we propose a hierarchical version of timed automata. This extension of timed automata ultimately aims at verifying UML state-charts. Finally to help engineers to "see" if their model is correct and to make the simulation more interactive, an animator based on hybrid automata has been developed. This animator interacts tightly with timed automata and allows user generated events during the simulation.

Case Study: Modeling and Analysis of a Field-bus Protocol

In this study we report on an industrial application of the UPPAAL tool to model and debug a commercial field bus communication protocol. This protocol is developed and implemented for safety-critical application, e.g. process control. It has been running in various industrial environments over the world for the past ten years. During its seven years on the market, a number of errors have been detected, which result in time–outs and retransmissions. Due to the complexity it has been very time and resource consuming to troubleshoot these errors.

The company's interest is to improve the development process, reduce the maintenance time/costs and to improve quality of the product with the help of formal methods. The goal of the project is not to verify the correctness of the protocol in any sense of *completeness*, which is basically impossible due to the size and complexity of the system, but to localize the error sources in both the protocol logic and the implementation at the source level.

To our knowledge, this case study is the largest reported so far, where the UPPAAL tool has been applied. The whole protocol involves hundreds of pages of protocol specification and more than 27000 lines of source code. The study was carried out on the core of the protocol, which involved 151 pages of documentation and 5541 lines of Modula–2[1], which pushed UPPAAL to its limits. We show to which extent an academic tool can be used in practice in an industrial context.

We adopt an engineering approach to achieve our primary goal, which is to find bugs. The protocol is divided into 2 parts to tackle its size and complexity. The larger models are built incrementally on top of simplified models studied separately. It is an engineering approach since it is very much related to the components used in industry.

During the case study, a number of errors in the protocol logic and its implementation have been found and debugged based on abstract models of the protocol; respective improvements have been suggested. It turns out that many of the problems are due to incorrect usage of synchronization and timing mechanisms in the implementation of the protocol, in particular, semaphores and time–outs.

[1] figures obtained with `wc`

Timed Automata Extension: From HUppaal to Uppaal

We propose a hierarchical extension of timed automata. This formalism is meant to be close to UML state-charts to ultimately allow its formal verification. This work is carried out in connection with the WOODDES [2] project, which aims at improving design process, methods, and tools for real-time embedded systems. We are particularly involved in the state-chart diagrams. We propose a rich extension of Uppaal to meet this goal.

A number of modeling and verification tools for real-time systems have been developed based on the theory of timed automata. They have been successfully applied in various case studies. They have mainly been used in academic community and they become to enter industry, though still as academic products. On the other hand the commercially available tools offer design capabilities [Rha, HG97, Vis] with simulation while verification is limited. Some of them offer strong proof capabilities but are weaker on the modeling side [pro].

The state-charts formalism is appreciated by engineers because it is intuitive and graphical. The verification part is usable because it is automated and error traces are generated to allow graphical debugging of systems. Hierarchical models are concise which allow complex systems to be handled. The challenge lies in the model-checking to limit the explosion due to this conciseness.

Though we have a working prototype for a grammar and a translation, it is to be considered work in progress. We made the implementation accessible for future reference as frozen version at `http://www.brics.dk/~omoeller/hta/vanilla-1/`.

The translation version-ed `Vanilla-1` is documented as a milestone to make experiments with. It is not able to translate some powerful modeling constructs, though they are already present syntactically.

Unresolved Issues in Vanilla-1 are in particular local declarations, scope overriding, history entries, synchronization mechanisms other than handshake communication, and parameterized templates.

In near future, it is planed to implement an editor for the hierarchical grammar in the Uppaal tool. Simulation and verification of hierarchical models, however, are done on flat Uppaal timed automata, constructed by future versions of Vanilla-1.

There is a strong correspondence between hierarchical and flat traces. However, the imperative of introducing fresh and unambiguous names for flattened constructs makes it difficult for a human user to see this immediately. One possible remedy for this is to equip the Uppaal simulator with the appropriate mapping, so it can display names as specified in the hierarchical system. We feel that it is also necessary to provide a translation of TCTL formulas to corresponding ones in the flattened version. This seems to be purely syntactical, but strongly dependent on the mapping of local and global variables.

Looking ahead, we believe that there is a great potential for exploiting the hierarchical structure directly in terms of shaping more efficient model checking algorithms.

A Real-Time Animator for Hybrid Systems

Uppaal is a software tool for modeling, simulation and verification of real time systems that can be described as timed automata. In recent years, it has been applied in a number of case studies [KrJKW, LPY98, LP97, HSLL97, RT], which demonstrates the potential application areas of the tool. It suits best the class of systems that contain only discrete components with real time clocks. But it can not handle hybrid systems, which has been a serious restriction on many industrial applications. This work is to extend the Uppaal tool with features for modeling and simulation of hybrid systems. This part is based on the paper [ADY00].

A hybrid system is a dynamical system that may contain both discrete and continuous components whose behavior follows physical laws [Hen], e.g. process control and automotive systems. In this paper, we shall adopt hybrid automata as a basic model for such systems. A hybrid automaton

[2] Workshop for Object Oriented Design and Development of Embedded Systems

is a finite automaton extended with differential equations assigned to control nodes, describing the physical laws. Timed automata [AD94] can be seen as special class of hybrid automata with the equation $\dot{x} = 1$ for all clocks x. We shall present an operational semantics for hybrid automata with dense time and its discrete version for a given time granularity. The discrete semantics of a hybrid system shall be considered as an approximation of the continuous behavior of the system, corresponding to sampling in control theory.

We have developed a real time animator for hybrid systems based on the discrete semantics. It can be used to simulate the dynamical behavior of a hybrid system in a real time manner. The animator implements the discrete semantics for a given automaton and sampling period, using the differential equation solver CVODE. Currently the engine of the animator has been implemented in Java and C using CVODE. We are aiming at a graphical user interface for editing and showing moving graphical objects and plotting curves. The graphical objects act on the screen according to physical laws described as differential equations and synchronize with controllers described as timed automata in UPPAAL.

Contents

List of Figures

List of Tables

Part I

Modeling and Analysis of a Field Bus Protocol

Chapter 1

The Field Bus Protocol

1.1 Overview

This field bus protocol is designed for 80 stations communicating over a bus. Figure 1.1 shows at which network level this protocol is used. The protocol is used for process control.

A station acting as a "master" may initiate a dialog with up to 79 other stations acting as "slaves" in this dialog. The master requests information from a slave that only responds to it, thus the names master and slave. In fact the dialog is established between applications on stations. Each station may have several applications running, acting as masters or slaves.

The protocol has two main layers that are the field interface (FI) to access the protocol from the application, and the bus coupler layer to access the bus from the FI layer. These layers correspond respectively to the transport and data link layers in the OSI protocol standard [Tan81]. There is actually another layer below called the bus queue that is taken into account in the study for its capacity to store messages, i.e. to generate delays. Figure 1.2 depicts the protocol stack. The field interface (FI) covers the service data transfer and partly the message transfer layers. The bus coupler covers mainly the message transfer protocol and partly the the packet transport protocol. The bus queue is on the packet transport protocol.

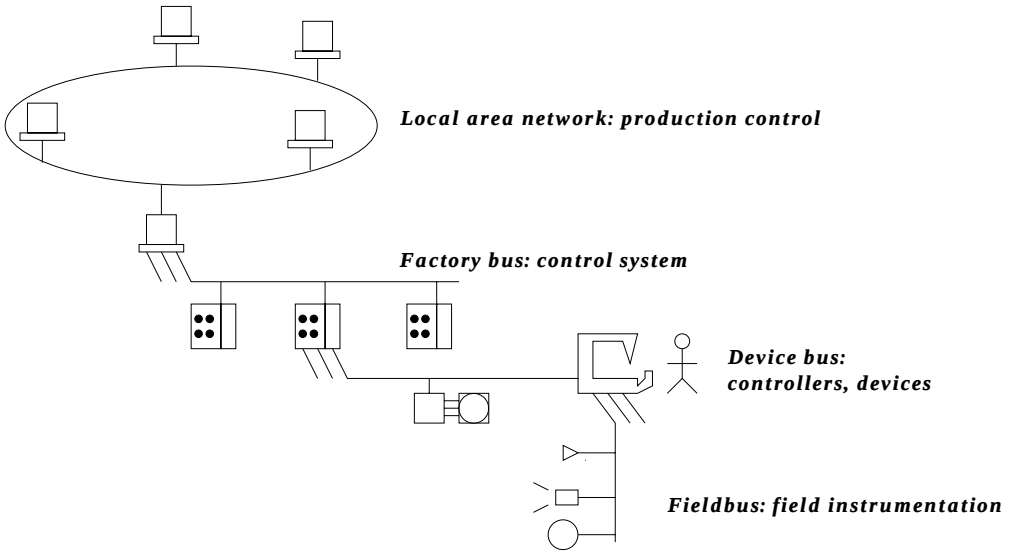


Figure 1.1: Factory communication networks

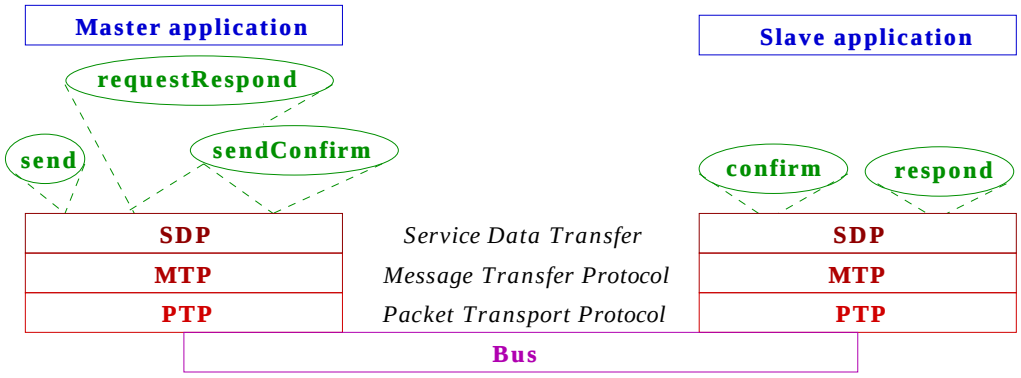


Figure 1.2: Protocol layers.

A typical scenario is as follows: a client application uses the master part of the FI to send requests to another station where a server application will respond through the slave part of the FI. Figure 1.3 shows an overview with four stations over a bus. Each of them has the described layers: the application, the FI, the bus coupler and the bus queue.

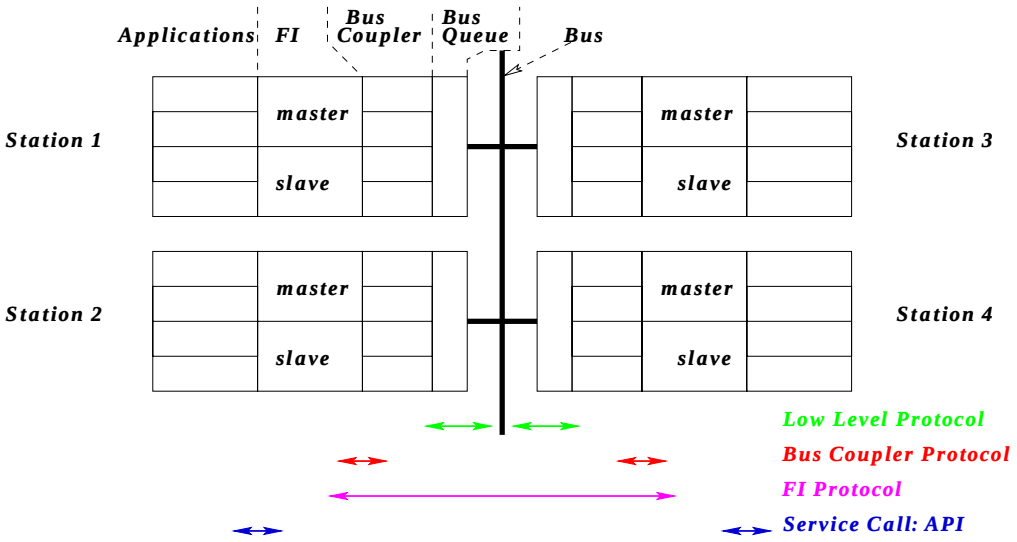


Figure 1.3: An overview of the protocol.

The different layers communicate with a specific protocol. We are interested in the bus coupler and in the FI protocols. The bus coupler protocol is the communication between the FI layer and the bus coupler layer. The FI protocol is the communication between the two FI entities on two stations. The low–level protocols for the communication with the bus and between the two bus coupler entities are not studied.

1.2 Field Interface : The Transport Layer

1.2.1 The Structure

This interface provides the services depicted in figure 1.2 to the application running on the station. From the master point of view, the services are of two types: the *sendConfirm/requestResponse*

and the *sendMessage*. The first one waits for an answer coming from the slave. The answer may be simple for a confirm of type yes/no, or more complex for a full response. The slave part has the corresponding services to answer when necessary. Note that the slave will act like a server and the master like a client. Message passing through the protocol, depicted in figure 1.3, is as follows:

1. the master sends a message to the field interface
2. the field interface (master side) decomposes the message into packets and sends them to the bus coupler
3. the bus coupler (master side) sends the packets to the next bus coupler via the bus
4. the next bus coupler (slave side) sends the packets to the field interface
5. the field interface (slave side) receives the packets, rebuilds the message and sends it to the application that is waiting on a signal

In addition, an acknowledgment mechanism ensures at every interface that messages are transmitted correctly.

The field interface has three main parts, both for the master and the slave:

- the *application programming interface* (API). For the master, it is the different send functions *sendMessage, requestResponse, sendConfirm*, and the *receive* function call that will block until answer arrives or a time–out occurs. For the slave it is the equivalent *receive* function call and the *sendMessage* function that sends a *confirm* or a *response*[1].
- the *packet time–out supervisor* that monitors time–outs. The master and the slave are monitored. This is a task that wakes up periodically to decrement and check a time–out counter. When a time–out occurs, the global state variable of the master or the slave can be changed. A time–out may be (re)set by another task, which is viewed as a normal time–out by the supervisor.
- the *receiver task* that is *MasterReceive* for the master and *SlaveDispatcher* for the slave. This task runs separately, listening to the bus coupler, assembling messages. When a message is ready, it puts it into a queue and *signals* a semaphore. As we are not interested in the message itself, only the semaphore is modeled. The different "receive" functions do a *wait* call on this semaphore.

The master and the slave have a state machine each that describes how they should behave. These correspond to the specification of the behavior of the master and the slave. The slave has three states and the master five. These states are global states that can be modified by different tasks. Priority and mutual exclusion are used to keep consistency and the study focuses on these global states.

So in addition to the control tasks corresponding to the protocol, both the master and the slave have a monitoring process that accepts all transitions between these states since the implementation just assigns the state variable. The monitoring process checks valid transitions and detects bad ones. This is a way to check that the implementation follows the specification.

Figure 1.4 shows the tasks involved in the field interface and the layer organisation. The left hand side of the figure depicts a master station communicating with a slave station (right hand side). In a client/server scheme the master acts as the client and the slave as the server. The flow of control of the client application goes to the functions of the field interface API. Other tasks are involved, but not at this level. The same applies for the server.

The protocol reserves four ports for the communication. Ports 1 and 2 are reserved for the master part and ports 3 and 4 for the slave parts. The communications port 1 to port 4 and

[1] The difference between these two is minor. The confirm is used for an answer of type yes/no and the response for more detailed information. The other technical differences are out of scope for this study.

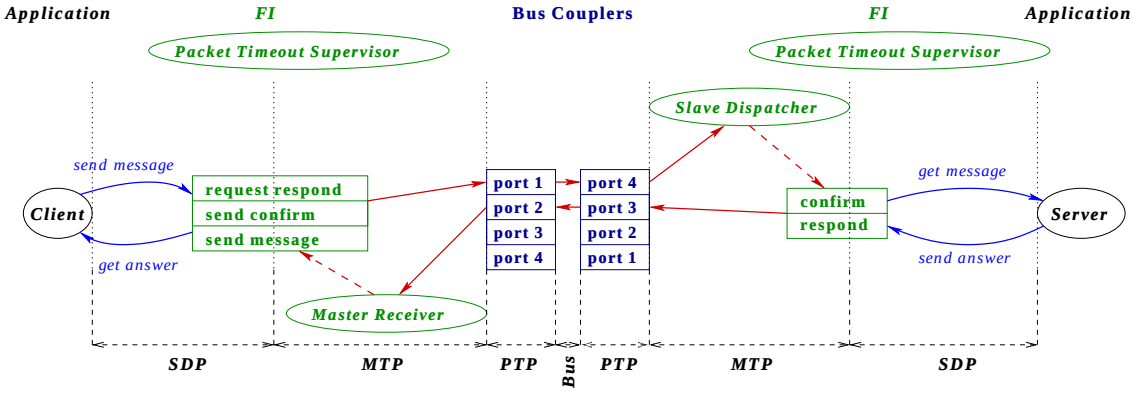


Figure 1.4: Tasks of the FI with respect to the layers. Real tasks are represented as circles, functions as rectangles.

port 3 to port 2 are identical and concern the bus coupler and lower layers. A bus coupler task is associated to each of these ports. The ports are used for communication between the bus coupler entities over the bus.

1.2.2 The Protocol Logic

The field interface protocol concerns communication between two applications, i.e. one master and one slave. It is an alternating bit protocol at the message level and a sliding window protocol at the packet level.

At the message level, a message bit is used as in the alternating bit, validating or not the messages to detect a missing message in the sequence. This is standard, though at a high layer. At the packet level, there is a very particular and important bit, namely the *transparent* bit. The sliding protocol works as follows: the first and the last packets of a window slide are not transparent and require acknowledgments. The packets in–between do not require acknowledgment, which is, real acknowledgment from the slave. However the bus coupler uses this bit to "lie" to the FI to deliver acknowledgments if the packet is transparent, or waits for a real acknowledgment from the slave if it is not. This is called a shortcut for efficiency in the design of the protocol since acknowledgments are avoided.

If there is an error, this will be detected at the end of the window. In this case the whole slide is retransmitted and the window size is reduced by one to adapt to transmission errors.

The modeling of the protocol concerns the tasks described in figure 1.4 that implement the send/receive of this sliding window protocol. Figures 1.5 and 1.6 show the state machines described in the documentation. They specify the protocol at a high level, i.e. retransmissions and sliding windows are not mentioned. The master is used this way: it is normally in the *dormant* state and it goes to the *awaiting first packet* state when a request is transmitted and it waits for the first packet of the answer. It goes to the state *receiving* while receiving the answer and then back to *dormant*. The slave has to answer requests from the master, so it waits in the state *idle (I)* or *idle after error (IAE)* depending on previous errors. If a multiple–packet message is received it goes to the *active (A)* state to receive the whole message and when the last one comes in the slave goes to the state *wait for receive task (WFR)* to wait for the answer from the master. The answer is processed in the state *answer outstanding (AO)* to know if an error occurred or not.

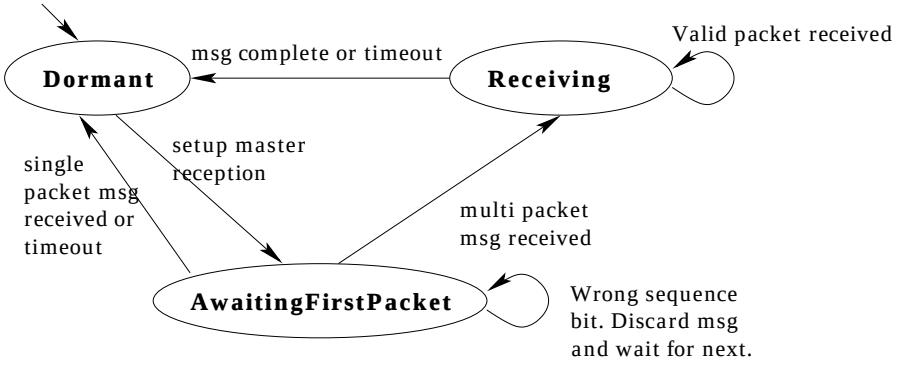


Figure 1.5: Master protocol state machine specification.

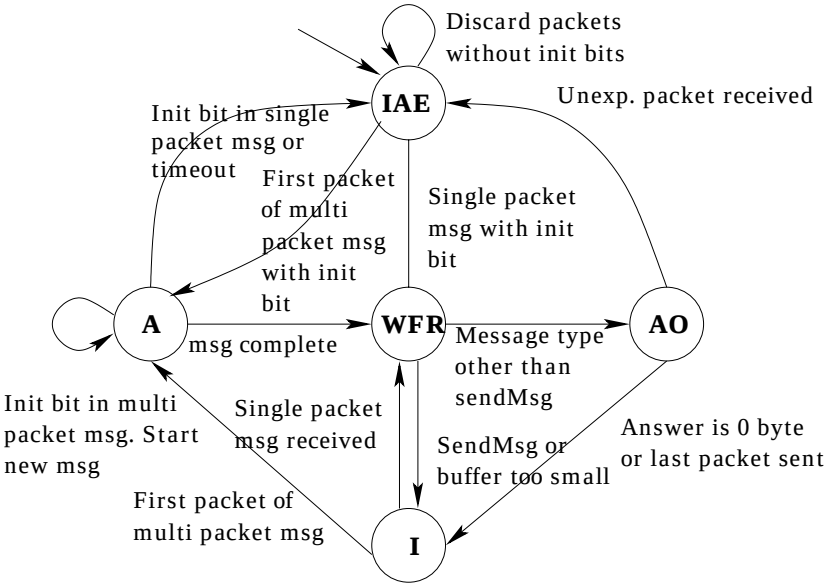


Figure 1.6: Slave protocol state machine specification.

1.3 Bus Coupler : The Data Link Layer

1.3.1 The Structure

As mentioned in the previous section, the bus coupler is the layer below the field interface. The tasks of the bus coupler run on a different board than the tasks of the FI. The operating systems are different as well. The design is motivated by the fact that this protocol is implemented on different hardwares and the lower–level layers can therefore be changed. This is for flexibility purposes.

The bus coupler corresponds to the data link layer in the ISO standard. It communicates with the FI via an interface, that is a shared buffer. Each bus coupler entity serves a port that is used by the field interface, that is the different FI entities communicate with each other though some ports, via the bus coupler that makes the link. These ports are used in the following way:

- a *request* from the master is sent to *port 1*
- this *request* is received by the slave from *port 4*

- a response is from the slave is sent to *port 3*
- this response is received by the master from *port 2*

Ports 1 and 2 are dedicated to the master and ports 3 and 4 to the slave. The ports are used to define the two communication channels port 1 → port 3 and port 4 → port 2. These two are identical from the bus coupler point of view. This means that the tasks serving ports 1 and port 4, port 3 and port 2 are respectively identical. The bus coupler model takes into account only one of these symmetrical communications, that is master sends via port 1 to slave receiving via port 4.

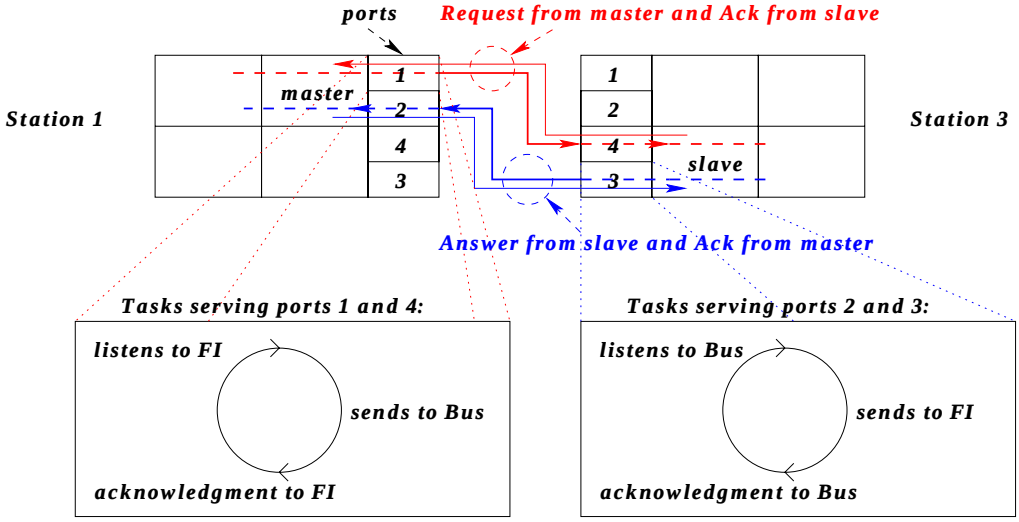


Figure 1.7: Bus coupler communication scheme with the different tasks.

Figure 1.7 illustrates this communication scheme and the tasks that we are focusing on. Figure 1.4 shows the same communication, though from the FI point of view. The tasks serving the ports 1 and 4 are symmetrical in the sense that they perform identical tasks, only the receiver and sender being switched.

The task serving port 1 listens to the FI. When it gets a packet, it forwards it to the bus and acknowledges the FI to notify that the packet was sent. Keep in mind the role of the transparent bit at this stage: if the packet is transparent, the acknowledgment will be positive (ACK). If the packet is not transparent the bus coupler has to wait for a real acknowledgment from the other side. Depending on the answer it will acknowledge the FI positively (ACK) or negatively (NACK).

The task serving port 4 listens to the bus. When it gets a packet, it forwards it to the FI and waits for an acknowledgment from the FI. If the packet is transparent, the acknowledgment is always positive. It is very similar to the other task.

Communication between the FI and the bus coupler entities at a port is achieved via a buffer. This buffer is separated into fields writable by only one side, but readable by the other one. The synchronization is based on these bits and a signal mechanism that uses interrupts through the two operating systems. The different bits used for synchronization are:

- *mail box reserved* to access the buffer
- *data read* to notify that data has been read
- *data written* to notify that data has been written
- *data lost* to return positive or negative acknowledgment

From the FI, these bits have in practice slightly different names, which is unfortunate. We will only consider the bus coupler view. In addition to these bits, a data field for the useful information is used. The FI side is referred as *cpu* because it is the application side at a high level. The bus coupler is referred as *dev* because it is the device side with low level communication with the bus. In practice the FI requests an interrupt to the OS. The OS notifies the other board, which generates an interrupt to the other OS. The interrupt is handled by an interrupt handler that signals a semaphore. The concerned task is waiting on that semaphore. The model considers only the semaphore.

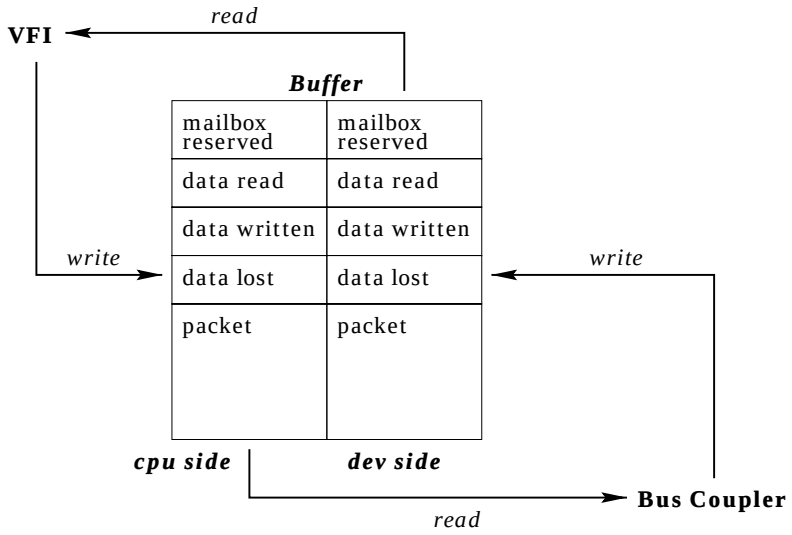


Figure 1.8: The configuration of the interface between the bus coupler and the FI.

Figure 1.8 illustrates the configuration of the buffer interface.

1.3.2 The Protocol Logic

The bus coupler protocol has two main parts: the communication with the FI through the buffer interface and the communication with the other bus coupler. The communication between bus couplers involves a minimum control of packets with management of re–sending packets and associated acknowledgments. The layer below is used via a simple API with send and receive primitives. This part of the protocol is known to be robust so we will not treat it in this study. Figure 10.1 in appendix 10 shows the protocol we are interested in, namely the communication with the FI. Note that when waiting for a bit, a time–out value is specified and a time–out result can be returned, leading to a reset and another try.

The implementation of the protocol uses signals to notify the reading side when a bit has been written. The mechanism with interrupts and signals is specific to this implementation and uses semaphores on both sides (different boards with their own OS each) and the modeling stresses this feature.

Chapter 2

Modeling and Abstraction

2.1 The Modeling Process

We adopt a top–down approach first to find and understand the relevant components of the system and then a bottom–up approach with progressive abstractions that allows us to build up several abstract models for verification. At the beginning of the project it was not planned to model the bus coupler but it turned that this was necessary.

The following steps are taken in the modeling process as illustrated in figure 10.2 (appendix 10):

1. model the bus coupler, based on the source code. This gives the *detailed bus coupler model*.
2. simplify the bus coupler model with classical abstraction techniques.
3. model the FI master and slave sides separately, based on the source code.
4. derive tests for the master and the slave, combine bus coupler abstraction, master/slave test and the slave/master models.
5. validate results on the two partial models with the help of the complete master and slave model which contains the bus coupler abstraction.

Step 1 is to construct a detailed model based on the source code of the bus coupler, which is presented in sub–section 2.2. This model is called the "detailed bus coupler model". Step 2 is to derive an abstract model presented in sub–section 2.4. Abstraction techniques such as hiding and the abstraction features of UPPAAL are used. Step 3 is to construct detailed models of the master and the slave separately, based on the source code, in sub–sections 2.6, 2.7, 2.8. Step 4 is to derive test automaton that simulates outputs of these components (input is ignored). The generated messages follow the logic of the protocol and can send negative acknowledgments randomly. These test automata are used against the partial master and slave models. Step 5 is to validate properties that are not satisfied, that is, the counter examples found in the partial models are validated for the detailed model, sub–section 2.9.

Figure 2.1 shows with respect to the overview of figure 1.3 what is modeled in the two steps of the study. First the bus coupler is modeled (transparent ellipses) with abstractions of lower and upper layers. Second the FI is modeled with abstraction of the previously studied bus coupler and an abstraction of the application using the FI.

2.2 A Detailed Model of the Bus Coupler

This model is the detailed model from which the whole bus coupler study is carried out. It is very close to the code and it is possible to map the model to the code. It is of great interest for the

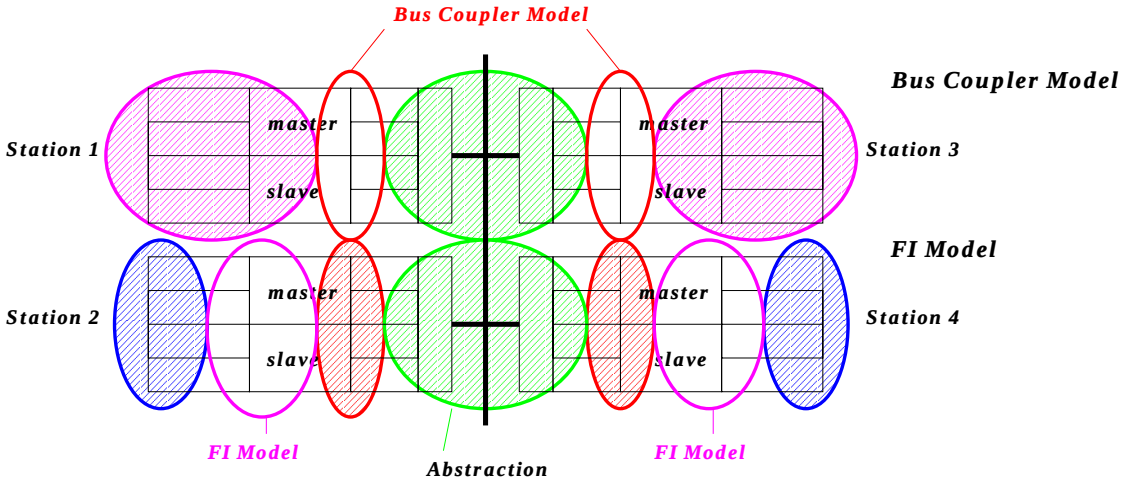


Figure 2.1: The two–steps modeling.

industrial partner that traces obtained from the verifier can be mapped to an execution trace of the real code for debugging purposes. To validate a trace is to validate the associated execution trace in the implementation.

The UPPAAL model of this part of the implementation consists of 14 automata, 4 clocks and 32 integer variables modeling 2 processes sending, 2 processes receiving, 4 semaphores and 6 functions. We consider here one master application on one station communicating with one slave application on another station. The structure of the model follows the description given in figure 1.7. The structure of the model is given in figure 10.3 (appendix 10). Circles denote processes and rectangles functions. The functions are modeled as processes in UPPAAL due to size consideration but they behave like functions.

The data content of the packets is irrelevant to the correct behaviour of the protocol since it is a layered protocol. The data carried by a layer has no meaning for this layer. There is however an exception, namely the transparent bit. This information bit is checked at the field interface and at the bus coupler layers, even though it is encapsulated as data for the bus coupler packets. This is due to the handling of transparent packets that do not require acknowledgments. In the model, this bit is the data of the packets. It may have the value -1 to mark corrupted data that should not be read, or 0/1 as valid values.

The stations have several applications, slaves or masters. They all use the same bus coupler layer serving the ports 1-4. A mutual exclusion mechanism is used to get access to the bus coupler at the field interface level. This is not modeled here. At the bus coupler layer, the incoming packets have a random sequence of 0, 1 or -1 since accesses may come from several applications in different states. This is modeled by non–deterministic choices.

The same idea is used in the receiver part where the upper layer may accept or reject a packet, which is positive or negative acknowledgment. These acknowledgments are non–deterministic since the FI is abstracted. Note that -1 is not a value part of the protocol but used for verification purposes.

The model consists of two bus couplers, one on the master side and one on the slave side. They are connected in practice via a lower layer to the bus. The communication protocol has another low-level that is partly modeled. The bus is modeled as a lossy channel preserving the ordering of data packets. There is a bus queue on each side of the bus, between the bus coupler and the actual bus. The queue only introduces delay when a message is to be sent. The delay varies if the queue is full or not, which depends on the accesses done by several applications and the traffic. This is therefore random in the model. The model for sending to the bus is a non-deterministic sending within a time window. Time–out may occur. Transmission delays are neglected with respect to

the time–out values controlling the protocol.

2.3 Abstraction Techniques

In addition to the UPPAAL abstraction mechanisms, we apply two kinds of abstractions on the detailed model to study different variations of it and to derive a simpler model without implementation details.

2.3.1 Abstraction Mechanisms in UPPAAL

In the modeling process, the models are refined by various modeling mechanisms implemented in UPPAAL including:

- *committed* states: the state must be left immediately with no delay. Interleaving is allowed only between the committed states. Atomicity in a sequence of states may be achieved, thus reducing the state space. The main goal of committed states is to reduce explicitly interleaving.
- *urgent* states: time is not allowed to progress in such a state, but all interleavings are allowed. It is useful to model race condition and non-determinism.
- *urgent* transitions: they should be taken whenever the guards become true. It is useful to model progress.
- shared variables: they are set and read atomically by processes running concurrently[1], thus suppressing the need for explicit mutual exclusion on them.

The goal of this low level abstraction is to control the level of non–determinism of the model.

2.3.2 Error Pruning

We study the detailed model in four different variants. The variations express different levels of assumptions on the program. The idea of the study is to use an *error pruning* technique, which is to detect an error but to stop exploration of the state space from the detected error state. This is done by causing a deadlock when this kind of error state is detected. It is important to notice that the deadlock in this case is part of the model only, not the protocol, and is used only for studying the protocol. We call this state of states the *error border*. The interpretation of the verification is as follows: if such a state is reached then the property is *partially* verified for a system that does not contain the "error" states. However we know that an error occurs; so we make another model with less pruning. Thus we have different refinement levels of the model with different levels of assumptions with associated partial results. This is useful to track bugs. The different variations are:

1. semaphore counters limited to 1, pruning error space
2. semaphore counters limited to 2, pruning error space
3. semaphore counters limited to 3, full space
4. semaphore counters limited to 3, checks added to correct the model

The limitation on the counter is still kept to bound state space generation. In the modeling process it was proved at a stage that one semaphore behaved badly, i.e. its counter grew beyond 3. It turned out that the model was not accurate enough and did not filter the synchronization properly. The model was refined and this behaviour disappeared. The limit of 3 comes from these

[1] on one transition

experiments where the goal was to include the case where one semaphore is at 3 and the others at 2. The corrected version is a modification of the original model, to patch the implementation. The models are constructed so that the following inclusions between the state spaces hold:

$$space_1 \subseteq space_2 \subseteq space_3$$
$$space_{1\setminus EB} \subseteq space_{2\setminus EB} \subseteq space_4 \subseteq space_3$$

Where $space_{i\setminus EB}$ denotes $space_i$ excluding the error border EB. The experiments in section 3.2 are consistent with these inclusions. These state spaces are comparable because $space_i \subseteq space_j$ comes from the fact that model j is a relaxed version of model i. The error border is meant to detect some states and it cuts the state space from these states. Removing these detection states and allowing further exploration gives the natural inclusions with $space_{i\setminus EB}$.

The corrections of model 4 concern checking bits when a signal is received. This is actually done in the function *receive* from the bus coupler side, but not on the FI side.

More formally, the error pruning technique used allows to partition the state space. Considering one semaphore, the values taken into account in the model form 3 classes: $[0][1][2 \ldots \infty]$. The model actually takes the semaphore into account up to 3. This is enough since the values of the variables and the clock regions are the same if there is a loop that makes the counter grow.

2.3.3 Hiding

To debug the protocol logic, we simplify the detailed model (which is based on the source code) using abstraction techniques and the modeling mechanisms listed above, in particular, the notions of *committed* and *urgent* states. The derivation of the abstract models takes away specific parts related to implementation which are the signal implementation and the way to wait on the bits, that is the interrupt handling and the semaphore management to signal a write or to ensure mutual exclusion. The implementation uses the sequence $interruption \rightarrow OS \rightarrow port \rightarrow interruption \rightarrow OS \rightarrow interrupthandler \rightarrow semaphore$. The abstraction allows a direct write/wait/read synchronization mechanism without semaphore, with the help of urgent synchronizations. The abstraction is independent from the implementation in the sense that this synchronization may be implemented in a different way.

We therefore hide all the semaphores and the corresponding variables. In addition to this we hide intermediate variables, i.e. result variables. This has the effect to collapse states and reduce the number of processes.

Table 2.1 compares the detailed bus coupler and the abstract version.

	Full	Abstract
Variables	32	15
Clocks	4	4
Processes	16	4
Locations^2	5.8e11	11520

Table 2.1: Comparison of detailed/abstract models.

2.3.4 Atomicity and Delays

As for the detailed bus coupler model we use different variants of the model to study the behaviour. The variations of the model are in two dimensions: breaking atomicity of transitions and allowing delay in reading bits. We obtain 5 models:

Model 1 is the simplest model where some transitions are considered to be atomic to study their consequences.

Model 2 relaxes model 1, by removing the atomicity of the transitions performing data-reading.

Model 3 relaxes model 2 by allowing delays when a bit is set to the expected value.

Model 4 also relaxes model 2 but by converting committed states related to data reading and writing to urgent states.

Model 5 relaxes model 4 by allowing delays as in model 3.

The models that are not relaxed do not allow delay when waiting on a bit to be set or reset. This is achieved in UPPAAL by an urgent synchronization that is always enabled, but in order to take the transition, the guard (the bit the component is waiting for) must be true. When relaxing the models, i.e. enabling delay, this synchronization is removed, allowing time to progress even if the guard is true. This models *eager* or *lazy* synchronization.

Atomicity in a sequence of transitions is modeled with UPPAAL *committed* locations. Locations that do not consume time but still do not have atomic transitions are marked *urgent*.

The models are derived so that the following inclusions hold:

$$\begin{array}{ccccc} space_1 & \subseteq & space_2 & \subseteq & space_4 \\ & & \cap| & & \cap| \\ & & space_3 & \subseteq & space_5 \end{array}$$

The idea is to derive models 3 and 5 to stress delay and models 4 and 5 to stress race condition. These inclusions hold by construction of the automata, which is, $space_i \subseteq space_j$ because model j relaxes model i by adding delays, or allowing interleavings (commit to urgent) *without* disabling the previous transitions. The basic model is the same, the set of variables is the same but there are more reachable states. The verification results are consistent with these inclusions.

2.3.5 Refining the Models

Refinement of the models is the opposite of abstraction. It gives more details and adds accuracy to the models. It is a trade-off since abstraction is needed to reduce state explosion and refinement is needed for accuracy of the model. In the study both were used side by side and refinement concerning the semaphore mechanism was used when a trace was not judged valid by the engineers.

2.4 Abstract Models of the Bus Coupler

The UPPAAL templates for the model 5 are given in figures 10.4, 10.5, 10.6, and 10.7 (appendix 10). The template automata for the other variants are similar with the described variations.

These automata are close to the protocol description given in figure 10.1. We recognize the master waiting for `devmbr==0` and `devdataR==1` with possible time–outs, corresponding to `wait(dev.mailbox==0` and `wait(dev.dataread==1)` on the figure. The random transparent bit of the packets, as explained previously, is modeled by non-deterministic transitions. States marked with `u` are urgent and those marked with `c` are committed.

The master bus coupler is the counter part of the master. It forwards data to the slave bus coupler and waits for an acknowledgment (non-transparent packets). Time-outs are possible here as well.

The slave counter-part receives packets from the master bus coupler and it can choose to send back acknowledgments (positive or negative) or not at all.

The slave receives from the bus coupler, it acknowledges positively or negatively. The waiting is similar to the first coupler: wait for `devdataW==1` and then for `devdataW==0`.

2.5 Relating the Models

2.5.1 Error Localisation

We are interested in properties at an abstract level from which we can infer conclusions on the concrete model. The idea is as follows : the concrete model is a particular implementation of a

protocol, the reduced model is the protocol itself where particular implementation details have been abstracted away. We use the term *reduction* that is more appropriate for our purposes.

The inference rule that we are looking for is :

$$\frac{R(M) \models \phi}{M \models \phi}$$

where M is the original model, $R(M)$ the reduced one, ϕ a formula of the form $\exists \diamond p$.

The point is to know if the protocol is correct without considering the implementation, but if something wrong is found then it is wrong in the implementation. This is a debugging process and our approach allows us to localise errors isolated from the implementation.

The reduction relation used here is the simulation : $R \rhd M$ the reduced model R can be simulated by the concrete model M, with respect to observable actions. The inference rule comes from this. The definition is:

Definition 1 (Simulation) $P \rhd Q$ if for all actions $\alpha \in Action$ whenever $P \xrightarrow{\alpha} P'$, then for some Q', $Q \xrightarrow{\hat{\alpha}} Q'$ and $P' \rhd Q'$. $\square$

We have to define now the reduction relation $\mathcal{R}_r$ that R has to satisfy.

2.5.2 Reduction Relation

We wish to establish a relation allowing us to remove a component and hide non observable variables. The hiding part of the problem is a standard hiding operation. It includes collapsing states when transitions between them are not observable with respect to a set of hidden variables. Removing a component is more difficult and in our case it is restrictive.

The scheme to remove a process is as follows : we have three processes in a model M such that P_1 communicates with P_3 via P_2. P_2 is the implementation of the communication. The static relation $\mathcal{R}_r$ that $P_1|P_2|P_3$ satisfies is such that :

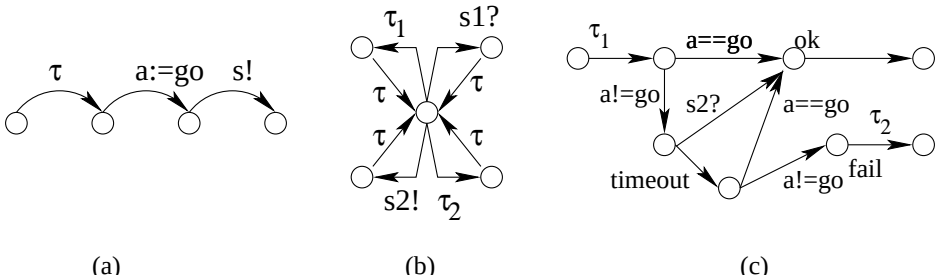


Figure 2.2: Complex automaton patterns.

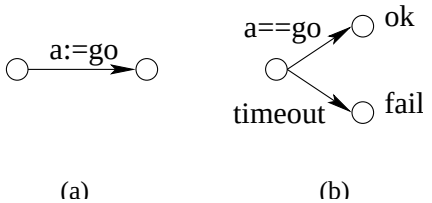


Figure 2.3: Reduced automaton patterns

- P_1 implements the behaviour shown in figure 2.2(a) where τ is an internal transition to prepare the communication, typically a reset ; $a := go$ sets the shared control variable a to the expected value go that P_3 is waiting for ; $s!$ is hand shaking synchronization.
- P_2 implements the behaviour shown in figure 2.2(b) where τ_1 can be related to τ in P_1 to prepare communication, and τ_2 could be a cleaning asked by P_3. The τ transitions are internal actions that ensure $P_3\ notified \implies P_1\ sent.$ $s_1?$ is hand shaking communication with P_1 and $s_2!$ with P_3.
- P_3 implements the behaviour shown in figure 2.2(c) where τ_1 is initialisation, typically clock reset, τ_2 post synchronization or cleaning actions with P_2.

These constraints are abstract and automata that simulate these satisfy $\mathcal{R}_r$ as well and they will clearly yield the expected property as well.

The reduced system is then P_1' with the reduced sub part figure 2.3(a) and P_3' with the reduced sub part 2.3(b). The transition labelled $a == go$ is urgent iff $s_1 \dots s_2$ is urgent. This reduction is in fact a busy waiting with time-out although the implementation is not, but this is the wanted behaviour of the protocol itself.

The result is then

$$\frac{\mathcal{R}_r(P_1, P_2, P_3) \quad R(P_1)|R(P_2)|R(P_3) \models \phi}{M \models \phi}$$

with ϕ of the form $\exists \diamond p$. We notice that $R(P_2)$ is empty.

Generalisation To generalise the idea of abstracting an implementation and extracting the underlying protocol logic that is implemented, the property becomes :

$$\frac{R(M) \rhd M \quad R(M) \models \exists \diamond p}{M \models \exists \diamond p}$$

This is straight forward but the point is to have the most relevant R as possible to keep interesting properties and our $\mathcal{R}_r$ allows us to reduce M to that interesting model which keeps the logic of the protocol. Other $\mathcal{R}_r'$ should verify $R' \rightsquigarrow M$. Such a relation is used for the FI models.

2.5.3 Relations between the Models

We have two basic models with variations in each. We note I_i the implementation models and R_i the reduced ones. As stated in section 2.3.2 :

$$I_{1\backslash EB} \subseteq I_{2\backslash EB} \subseteq I_4 \subseteq I_3$$

in term of space and from section 2.3.4 we have

$$\begin{array}{ccccc} R_1 & \subseteq & R_2 & \subseteq & R_4 \\ & & \shortmid\cap & & \shortmid\cap \\ & & R_3 & \subseteq & R_5 \end{array}$$

The relations are $R_1 \rhd R_2 \rhd I_1 \rhd I_2 \rhd I_3$. I_4 is not present because it does not contain the error states.

2.6 Modeling FI

The field interface model follows the protocol description given in figure 1.4. The master side has a sender process implementing the sliding window with transparent packet protocol. A receiver process listens to incoming packets and rebuilds messages that are responses from the slave. A

time–out supervisor process takes care of the master time–outs. A status process monitors the state transition for verification purposes. The master part has 3 main running processes and one monitor process.

The slave part is similar to the master part and has a sender, a receiver (called dispatcher) and a time–out supervisor as main processes. A state process monitors the state transitions here as well.

In addition to these 8 main processes, 2 semaphores, 1 mutex and one forward process are used.

The model implements only the service request response since it is the most complete and it contains the others. The sending parts of the model as well as the receiving parts (though the forwarder) contain an abstraction of the bus coupler depicted in figure 2.4.

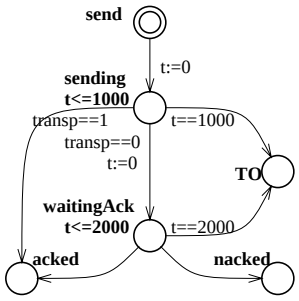


Figure 2.4: Bus coupler abstraction used.

The model components are depicted in figure 10.8 (appendix 10). The figure shows the communication between the components. As one guesses, the model is suitable to be cut into one master side and one slave side with a test in the place of the forwarder. The semaphores and the mutex are not depicted in the figure.

2.7 Detailed Models of FI Master

This model is verified with the components from the master side and the slave test. The sender automata are adapted at the bus coupler level because no real receiver is modeled. It is important to notice that these changes are very minor and consist of removing the channel synchronizations on the sending transitions. The test slave is depicted figure 10.9 (appendix 10). This test models the strict minimum of a slave station. It reacts to acknowledgments from the master, generates messages (replies) with correct sequence. This test automaton is derived from the slave sending function.

The master is monitored with the state monitor process depicted in figure 10.10 (appendix 10). This monitor has the three central states corresponding to the protocol states. Valid transitions are drawn directly between these states as described in the protocol. All the other undefined transitions are present as well, though they go through an error state (committed[3] in UPPAAL) to detect that the transition is taken. The specification of the master protocol state machine is given in figure 1.5. We recognize the three central states. The labels are the original ones.

The implementation has a state variable per application. This state variable is shared between all the components of the master, that run concurrently. It is important to keep the integrity of

[3] that is left immediately, but detectable for verification

this variable by a controlled access, which is done via mutual exclusion and means of preemptive scheduling. Reading or changing this variable is done directly by variable manipulation. The model defers to the state monitor when changing it. When setting the variable to a specific value, a channel synchronization for this specific value is used. The state monitor takes the transition corresponding to this assignment, does the assignment, and depending on defined conditions in the protocol, it will take a legal or an illegal transition. The monitor states have the particularity that the disjunction of all outgoing transitions is always true, to avoid artificial deadlocks. Reading is immediate and is always legal.

2.8 Detailed Models of FI Slave

The slave model is similar to the master model. It has a master test process generating messages as the slave test, though this one may change the init bit. The slave does not do this, but the real master may do this. The master test is depicted in figure 10.11 (appendix 10). This test automaton is derived from the master sending function.

The slave is monitored as the master by a state monitor process that works as the master's one. This monitor is depicted in figure 10.12. The specification of the slave protocol state machine is given in figure 1.6 (appendix 10). These states correspond to the five central states in the monitor automaton.

2.9 Validation of FI Models

The master and the slave models were verified against a test automaton. These models satisfy the simulation relation with the complete model, with respect to the observable events of the master, respectively slave part:

$M_{master} \rhd M_{complete}$	model with the master simulated by the complete model
$M_{slave} \rhd M_{complete}$	model with the slave simulated by the complete model

This holds with respect of the visible events in the master, respectively the slave. The events concerning actual sending of messages are hidden, though not the acknowledgments. This holds because of the way the tests were constructed. These tests are able to produce all the outputs of the hidden component.

However we are interested in the observation equivalence relation. Unfortunately the models are not equivalent since the tests are less constrained than the complete models. The point is to validate the models so that their behaviour is not too general with respect to the detailed models.

Experimentally, we rechecked the properties that were violated in the test models, i.e. those that gave counter–examples, to validate the counter–example in the detailed model. This is to ensure that the test behaviour does not deviate from the real behaviour.

Formally, we strengthen the simulation relation [Mil89] towards the observation equivalence relation with respect to a set of properties, which is, for a sub-set of real behaviours the observation equivalence relation holds.

Definition 2 (ϕ-Observation equivalence) $P \approx^{\phi} Q$ if

1) $P \models \phi$ and $Q \models \phi$,
2) for all $\alpha \in Act$:
   - i whenever $P \xrightarrow{\alpha} P'$ such that $P' \models \phi$, then for some Q', $Q \xrightarrow{\hat{\alpha}} Q'$ and $P' \approx^{\phi} Q'$ and $Q' \models \phi$,
   - ii whenever $Q \xrightarrow{\alpha} Q'$ such that $Q' \models \phi$, then for some P', $P \xrightarrow{\hat{\alpha}} P'$ and $Q' \approx^{\phi} P'$ and $P' \models \phi$. $\square$

In our case we have:

$$M_{master} \approx^{\phi_1} M_{complete}$$
$$M_{slave} \approx^{\phi_2} M_{complete}$$

This is confirmed by experimental results: formally, ϕ characterizes the sub-set of behaviors of the two models and on these two sub-sets are equivalent. In practice we have a set of properties verified or violated by both the abstract and the complete models. So experimentally, one could think that the models are equivalent, but by construction the abstract model is not limited in the generation of messages and can generate sequences that the complete model would not generate. In practice ϕ characterizes the valid traces of the abstract model and is not given explicitly. By executing the models and respecting these sequences, the models are equivalent with respect to these sequences.

Figure 2.5 shows the differences between trace equivalence, simulation and ϕ equivalence: X and Y are trace equivalent, they generate the same traces. X can simulate Y but Y can not simulate X: if Y takes the branch A-B-D, X has still the choice of E and C. The sub-tree verifying some ϕ are equivalent, as shown in a more abstract way in figure 2.6.

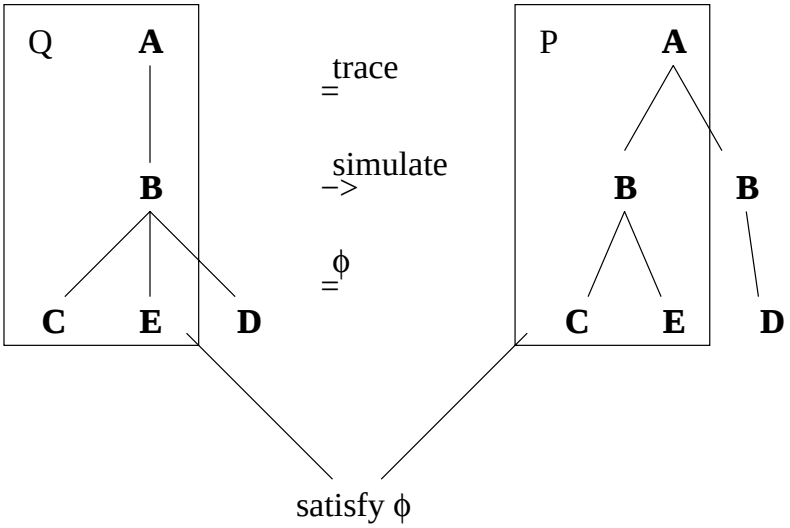


Figure 2.5: Trace equivalence, simulation and ϕ-equivalence.

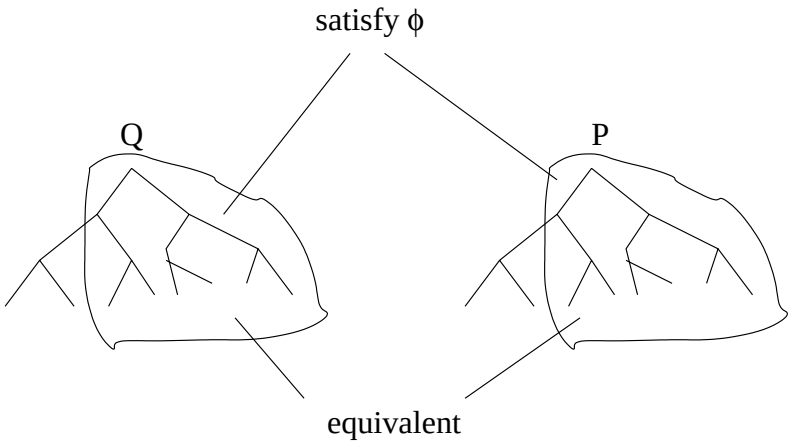


Figure 2.6: ϕ-equivalence.

Chapter 3

Verification

In this section we present the correctness properties checked. They are either reachability properties of the form $\exists\Diamond\ \phi$ or invariants of the form $\forall\Box\ \phi$. The predicate ϕ is defined over states, variables and time.

3.1 Properties

Finding the properties to check was a problem in itself because the documentation of the protocol was not adequate for verification purposes.

For the bus coupler models, 82 properties for the detailed models (and 35 for the reduced models) are checked. These properties are classified into 4 classes:

- 6 correctness properties for both the detailed and the reduced models related to the logics of the protocol
- 25 functional properties for the detailed models and 5 for the reduced ones, related to the synchronization of the components. Violating these properties could induce bad/wrong behaviour. The properties of the implementation models are classified as follows: 8 related to the implemented semaphores, 10 to detection of possibly bad states belonging to the *error border* and 7 related to precedence between states. The abstract models properties were based only on precedence.
- 19 behaviour properties for the detailed models and 5 for the reduced ones, which are intuitively believed to hold with respect to the protocol. This is expected behaviour which has only performance impact.
- 32 validation properties for the detailed models and 19 for the reduced ones, related to the model itself to validate it. The protocol works in practice and the model must work the same. A more complex model requires more validation hence the difference.

Concerning the field interface, 98 properties are checked. The classification is different: we have correctness properties and validation properties. There is no equivalent of the functional and behaviour properties as defined for the bus coupler. However the field interface has different priority tasks that are modeled. We check that this modeling is correct as well as the consistency between the state monitors and the state of the system.

- 32 correctness properties based on the state monitors.
- 50 simple validation properties related to the model itself to check that it works. These properties reflect the model of the implementation.

- 16 consistency properties related to the decoration of the model, i.e. parts of the models that are not originally part of the implementation. This checks that the priority model holds, as well as the consistency of the state monitors.

We do not intend to present all the properties but rather the important ones. Verification was conducted on a Sun Ultra-SPARC-II Enterprise 450 model 440, quadri processor 400MHz, though only one at a time is used by UPPAAL. The computer is equipped with 4GB of physical memory. UPPAAL version is 3.0.39[1]. Options were re-use state space, breadth-first search, active clock reduction and no trace generation to save memory[2].

3.2 Bus Coupler

3.2.1 Detailed Models

The resources consumed to verify the properties are given in table 3.1. These figures show the size of the complete state space because the properties need complete search. They are consistent with the inclusions $space_{1\backslash EB} \subseteq space_{2\backslash EB} \subseteq space_4 \subseteq space_3$ that were given in sub-section 2.3.2.Figure 3.1 illustrates the space inclusions.

Model	Size	Verification
1	129 MB	12:31 min
2	136 MB	14:41 min
3	149 MB	14:40 min
4	140 MB	11:11 min

Table 3.1: Resources used for verification.

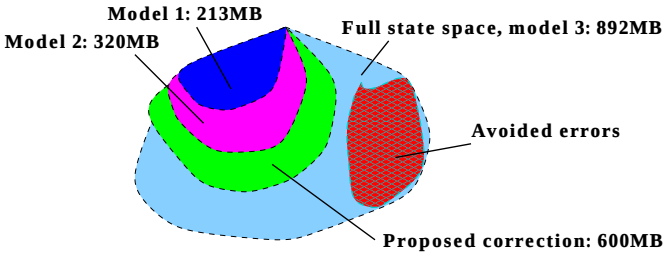


Figure 3.1: Overview of the state spaces and inclusions.

The correctness properties are:

```
1:  A[] VFIToCoupler_1P1.written imply vfiTrans1!=-1
2:  A[] (CouplerFromVFI_1P1.done and resultC11==0) imply bcTrans11!=-1
3:  A[] CouplerToBus_1P1.sent1 imply bcTrans11!=-1
4:  A[] CouplerFromBus_2P4.received imply bcTrans24!=-1
5:  A[] CouplerToVFI_2P4.step2w0 imply bcTrans24!=-1
6:  A[] VFIFromCoupler_2P4.dataTaken imply vfiTrans2!=-1
```

where `A[]` stands for $\forall\Box$. They concern the transparent bit (data modeled) which should not be written/read when not valid (-1) by the FI (vfiTrans) and the Bus Coupler (bcTrans). The full state model 3 does not satisfy property 6. More models used to violate more properties here

[1] distributed on the web
[2] exact options given to verifyta are: -CDSTaqs

but that was due to the granularity of the models themselves. Furthermore the trace of property 6 exploits an approximation of the model and is not judged valid by the engineers. Although the model fails to show a real error here, it shows that the components may reach a state where they are de–synchronized.

4 of the properties concerning semaphores are:

```
43:  A[] not SemVFItoCoupler24.signalNotTaken
44:  A[] not SemCouplertoVFI24.signalNotTaken
45:  A[] not SemVFItoCoupler11.signalNotTaken
46:  A[] not SemCouplertoVFI11.signalNotTaken
```

They mean that whenever a signal is sent, the previous one should have been accepted otherwise "it has not been taken". If there is a wait on that signal, it will not make much sense since the semaphore stores previous signals. These properties are not verified for model 1 but hold for models 2, 3, and 4. The counters may reach 2 but not 3, as pointed out in sub–section 2.3.2.

An interesting functional property (because of its result) is:

```
57:  A[] not VFIToCoupler_1P1.OKwhenMBR
```

It states that the FI side should not be in a success state after the first synchronization step if the mailbox receive flag is on (it should be off). This property is not satisfied by all the models. The interesting point here is that it is acknowledged by the engineers but it is not considered important because in this precise case, the communication concerns acknowledgment and no data. This is to be documented in the implementation. Other functional properties are very similar to this one.

2 precedence properties addressing synchronization are:

```
75:  A[] not (VFIToCoupler_1P1.testOK and (CouplerFromVFI_1P1.step1w0 or
CouplerFromVFI_1P1.step2))
76:  A[] not (CouplerToVFI_2P4.endWait2 and (VFIFromCoupler_2P4.waited or VFI
FromCoupler_2P4.wait0))
```

Only model 3 does not satisfy 76. Property 75 checks that one part should not be at the end of sending a packet with success while the other side still waits for acknowledgment. Property 76 checks that one part should not be sending an ackownledgment while the other part is going to begin to send a packet.

4 behaviour properties are:

```
10:  A[] not (VFIToCoupler_1P1.done and resultV1!=0 and bcTrans11==1)
16:  A[] not (Coupler_1P1.sentTO and bcTrans11==1)
33:  A[] not (Coupler_2P4.acking and saveTrans24==1)
78:  A[] (VFIToCoupler_1P1.testOK and vfiTrans1==1) imply devdatalost11==0
```

Property 10 states that sending a transparent packet should never fail and this is false for all models. Property 16 states that timeout should not occur on transparent packet which is true for all models. Property 33 states that acknowledgment is not sent after transparent packets which is true. Property 78 states that the coupler "lies" properly to the FI when a transparent packet is sent, which is true for all models.

Validity properties are simple reachability properties to check that the model does what it should do. One of these is: packets are transmitted successfully.

```
2: E<> VFIFromCoupler\_2P4.done and resultV2==0
```

The protocol is subject to de–synchronization, though it is not fatal. The origin comes from race conditions when reading from and writing to the buffer.

3.2.2 Abstract Models

The resources consumed to verify the properties are given in table 3.2. They are consistent with the inclusions

$$space_1 \subseteq space_2 \subseteq space_4 \subseteq space_5$$
$$space_2 \subseteq space_3 \subseteq space_5$$

referred in sub-section 2.3.4. 35 properties are verified.

Model	Size	Time
1	3.8 MB	8 sec
2	4.1 MB	9 sec
4	5.0 MB	10 sec
3	11 MB	32 sec
5	14 MB	37 sec

Table 3.2: Resources used for verification.

Due to the way we construct the models, we believe that $space_2 = space_3 \cap space_4$ though we can not prove it. Experiments confirm this by exhibiting this common behaviour with a number of properties. These different models are interesting when properties are verified in one model (case for $space_1$) but not in others ($space_2$, thus $space_3$ and $space_4$). This is used to pinpoint behaviour differences and see what the protocol is sensitive to.

The correctness properties are:

```
1:  A[] master.waitDataR imply vfitrans1!=-1
2:  A[] coupler1P1.sending imply bctrans11!=-1
3:  A[] coupler2P4.gotMsg imply store24!=-1
4:  A[] slave.read imply vfitrans2!=-1
5:  A[] master.OK imply devdatalost11!=-1
6:  A[] coupler2P4.readnottrans imply cpudatalost24!=-1
```

These are of the same type as the implementation properties. They state that wrong data should not be read because they are received too early or too late. We add here the explicit test on the acknowledgment answer from the coupler or the FI-slave with dev/cpudatalost. This is present in the implementation as well, but indirectly.

Properties 1 and 5 are satisfied by all the models. Properties 2 and 3 are satisfied only when no delay is allowed, which is the case for the models 1, 2 and 4. When delay is allowed a timeout may occur concurrently leading to an unwanted change that leads to a race condition. To interpret this as realistic or not, the hardware and runtime environment has to be taken into consideration. In our context of non-preemptive multitasking on the Bus Coupler side, this situation is possible if the coupler blocks while sending.

Property 4 is satisfied only for the first model. This property is sensitive to race condition. Property 6 is satisfied only for the 3 first models. Models 4 and 5 introduce new interleavings and a race condition is enabled by changing *commit* states to *urgent* states.

The functional properties are:

```
31:  A[] not (coupler2P4.readnottrans and slave.read)
32:  A[] not (coupler2P4.readtrans and slave.read)
33:  A[] not (master.OK and coupler1P1.sending)
34:  A[] not (coupler2P4.sending and cpumbr24==1 and slave.read)
35:  A[] not (master.waitMBR and devmbr11==1 and coupler1P1.sending and ck11==0)
```

They concern de-synchronization, when a component is one cycle late on the other. Properties 31 and 32 state that the coupler should not be in a state ready to read the acknowledgment from

the slave while this one has not written it and is about to do it: models 4 and 5 violate these properties. This result is similar to property 6.

Property 33 states that the master should not have read the acknowledgment from the coupler whereas this one has not written it yet. Model 5 does not satisfy this one, which means that this property is related to delay *and* race condition.

Property 34 states that the coupler should not be in a state waiting for the mailbox being available in order to write data while the slave has read data and not reserved yet the mailbox. This is satisfied by all the models.

Property 35 states that the coupler should not be in a state when it has just reserved the mailbox and read data from the master though this one is waiting for the mailbox to be freed in order to write data. Models 3 and 5 do not satisfy this one. This property is sensitive to delays.

The behaviour properties are:

```
26:  A[] not (master.timedout2 and vfitrans1==1)
27:  A[] not (coupler1P1.waitanswer and bctrans11==1)
28:  A[] not (coupler1P1.acking and ck11>0 and bctrans11==1)
29:  A[] not (slave.timedout2 and vfitrans2==1)
30:  A[] not (coupler2P4.timedout2 and bctrans24==1)
```

These properties are related to the nature of the packets: if they are transparent, timeout should not occur. This is an expected behaviour, but not a critical property. Properties 26 and 30 are not satisfied, which comes from a possible delay from the bus queue. The acknowledgment is sent to the master *after* having sent a message on the bus. If the queue is full and introduces delay so is the transparent packet delayed.

Property 27 is satisfied which is straightforward with respect to the automaton. Property 28 is not satisfied by models 3 and 5 which comes directly from the possible delays while reading bits. Property 29 is satisfied.

The conclusion on the abstract model is that the protocol is implementable since the first model is valid. However the implementation has to avoid some possible race conditions as well as some delays in order to work.

The bus coupler part does not show any major flaw. As it is a rather low level implementation, the models are very sensitive to the underlying modeling assumptions. However, the models proved to be useful with their identified limits. The consequence is a series of improvement requests on the implementation, i.e. the product will be improved as a result of the study.

3.3 Field Interface

The resources consumed to verify the properties are given in table 3.3.

Model	Size	Verification
Master, all properties	817M	1h 32min 4sec
Slave, all properties	1.46G	7h 34min 22sec
Complete, without satisfied safety properties	2.42G	6h 1min
Complete, all properties, bit state hashing	5.3M	2h 12min

Table 3.3: Resources used for the verification.

The method used was to validate the complete model with the simulator, check the master sub-model (against a slave test) with the master properties, similarly for the slave, and check the complete model with the validation properties and the violated safety properties. Furthermore the complete model is checked with the full set of properties, but with bit state hashing.

3.3.1 Master Model

The validation properties are of two kinds: reachability properties to check the functionality of the models. 2 of them are:

```
2:  E<> Master_Send.sendOK
4:  E<> Master_Send.NackTO
```

They check for the success of a sending and a time–out. These check the master. The other type is for the slave test process, there is only one, to check that a complete message is sent:

```
31:  E<> Slave.send and Slave.size==0
```

All these properties are satisfied.

The consistency properties are of type `A[]` ϕ to check priority handling (one property) in the model and the consistency of the state monitor (3 properties, one for each state):

```
10:  A[] (Master_Send.idle or Master_Send.sending1 or Master_Send.waiting1 or
Master_Send.waitReceive or Master_Send.sendingx or Master_Send.waitingx) imply MP4==0
20:  A[] ((Master_Status.D or Master_StatusDbadR) imply (MStatus==Dormant)) and
((MStatus==Dormant) imply Master_Status.D or Master_Status.DbadR))
```

All these properties are satisfied.

The safety properties of the state monitors check for bad transitions. In UPPAAL it is not possible to check for a transition so the trick is to use a committed state just for the detection. There are 9 of these. Two of them are:

```
14:  A[] not Master_Status.RbadAFP
18:  A[] not Master_Status.DbadR
```

These two properties are not satisfied.

We show unknown transitions may occur. Some of them were acknowledged by the engineers and they are investigating them.

3.3.2 Slave Model

The model–checking of the slave model is similar to the master model. The validation properties are of two kinds, the reachability ones, 2 of them are:

```
2:  E<> Slave_Send.sent1
3:  E<> Slave_Send.NackTO
```

They check for sending successfully one packet and getting one time–out. They are verified. The master test property is:

```
56:  E<> Master.send and Master.size==0
```

There are 5 consistency properties concerning the state monitor (one for each state), one of them is:

```
44:  A[] ((Slave_Status.AO or Slave_Status.AObadAO or Slave_Status.AObadWFR or
Slave_Status.AObadA) imply ((SStatus==AnswOuts)) and ((SStatus==AnswOuts) imply
(Slave_Status.AO or Slave_Status.AObadAO or Slave_Status.AObadWFR or Slave_Status.AObadA))
```

The consistency property concerning the priority is:

```
9:  A[] (Slave_Send.respond or Slave_Send.send0 or Slave_Send.sending1 of
Slave_Send.waiting1 or Slave_Send.idle or Slave_send.waitingMutex or Slave_Send.toI or
Slave_Send.sendingx or Slave_Send.waitingx) imply SP4==0
```

All these properties are satisfied.

There are 26 safety properties concerning the state monitor. Two of them are:

```
18:  A[] not Slave_Status.IAEbadI
26:  A[] not Slave_Status.AbadI
```

These two properties are not satisfied.

For the slave model we show here as well that unknown transitions may occur. Some of them were acknowledged and are under investigation.

3.3.3 Complete Model

The bit state hashing experiment on the complete model proved to be unsuccessful. It is an over–approximation method where positive results are not reliable whereas negative results are. Unfortunately the verification gave too many false positive, so it does not apply to our study.

The full verification of the carefully chosen properties was successful. We succeeded in proving that properties violated on the partial models were still violated on the detailed model. The detailed model satisfied also all the simple validation properties.

For our model and our set of properties the relation given in section 2.9 holds.

The conclusion for the FI part is that we identified unknown behaviours on both the slave and the master parts. The models are accurate enough to reproduce real code execution. Engineers are analysing these traces to improve the code.

Conclusion

This study is regarded as a success from the academic and the industrial point of view. For the academic side, we succeeded in modeling and analysing a real product, not a toy example. We applied a step by step method with abstractions and build incrementally our models on these abstractions. Although the technique used is still manual, it was feasible and it increased the knowledge of the code. Another aspect of the study is that we succeeded in teaching a formal method to industry. From the industrial side, it is a success since they will improve the code from a request for improvement of the code that is being written. They adopted the model–checking method, though what we did was reverse engineering. They will take a course on model–checking and they want to apply it from the design phase, before writing the code. The tool is better used this way.

Part II

Hierarchical Timed Automata

Chapter 4

Overview

4.1 Hierarchical Timed Automata

Hierarchical structures are a powerful mechanism to describe complex systems. They benefit from concepts like modularity and encapsulation and scale up well in industrial settings.

Modeling languages—like UML [BRJ98]—use hierarchical structures to organize design and specifications in different views of a system, meeting the needs of developers, customers, and implementors. In particular, they capture a notion of *correctness*, in terms of requirements the system has to meet. Formal methods typically address *model correctness*, for they operate on a (possibly very close) mathematical formalization of the model. This makes it possible to prevent errors inexpensively at early design stages. Of particular interests are state-chart-like models [Har87,HG97,HN96], that describe a behavioral view and allow execution of a model on a high level.

Our ambition is to build a hierarchical real-time formalism—called Hierarchical Timed Automata model or HTA model for short—, that can be used as input for model-checking tools. Correctness requirements are expressed in a dialect of the TCTL [HNSY94] logic. If we want to preserve decidability, this dictates restrictions on the expressiveness of the model. We need a formal definition of the *semantics* of this formalism to define the set of legal executions.

In the context of UML, this work aims at defining a real-time profile for UML state-charts, which is, a specialization of the general state-charts tuned for real-time applications. Use of UML as proposed in [Dou99] applied to real-time systems does not focus on state-charts and the specific real-time features are limited to time events. The reference method presented in [Dou00] to develop real-time systems with UML is focused on the tool Rhapsody and thus its features only. Furthermore the state-charts diagram is the main behavioural diagram of UML and its analysis is not addressed in these books, nor in the official OMG UML specification. Furthermore, specifications and properties of systems are described in terms of other diagrams, and more specifically message sequence charts (MSCs). This is in contrast with the use of timed automata and the TCTL formulae used to express formal properties.

UML is a standard and is used in the industry. There are few limited tools to check properties on the UML state-charts. UML state-charts have a lax semantics. The timed automata formalism is well understood and has proven its power in modeling and verification, though it is not hierarchical. These four facts are the motivation for our work: we extend timed automata with hierarchy and limited features of state-charts, as history, to obtain a specialized UML state-charts for real-time.

HTA are based on preliminary work done by Wang and David [DY00], refined by Moeller and David [DM01]. The hierarchical structure of the model comes directly from UML state-charts, though with strict restrictions concerning the entries and exits of super-states. The expressiveness is not affected and the models are more structured. It is more tedious for the user but a sufficiently smart editor can operate on a less restrictive input language, translated to the restrictive one. This

lax version of the state-charts is addressed because it is useful from a user point of view.

Analysis taking advantage of the hierarchical information is difficult to attain and this topic is beyond this licenciate thesis. However, in order to experiment with our formalism, we describe a translation of HTA to flat UPPAAL timed automata. The physical HTA representation is a XML document type definition. Models represented in this format are translated automatically to UPPAAL models and verified.

4.2 Informal Description

4.2.1 Elements of the HTA Structure

Hierarchical timed automata are basically hierarchical state machines, that can be put in parallel on various levels. The basic units of control are called *locations*, which may be *basic states* or *super-states*, i.e., itself a (hierarchical) state machine. In the latter case, the contained locations are called *sub-states*. At any point, a location is either *active* or *inactive*. Super-states can be of type *XOR* (where exactly one sub-state is active, if the super-state is active) or of type AND (where every sub-state is active if the super-state is active).

Transitions connect locations. If source or target of a transitions is a super-state, the transition connects to distinguished *entries* and *exits*. These are auxiliary structures and referred to as *pseudo-locations*. Pseudo-locations are different from ordinary (*proper*) locations, since they mark intermediated steps in a more complex transition, and cannot be part of a proper configurations, see below.

Transitions can be equipped with *guards*, *assignments* and at most one *synchronization*. Transitions are enabled, if their guards evaluate to **true** (in the current configuration), the synchronization (if any) is possible and they can reach a target configuration. A transition is not enabled, if taking it would lead to a configuration where a location invariant is violated.

As auxiliary constructs, *pseudo-transitions* ($\langle$*connector*$\rangle$s notation refers to the Element names in the XML grammar. At least one end of a pseudo-transition connects to a pseudo-location. Pseudo-transitions cannot be augmented with synchronizations, but in special situations may carry guards and/or assignments, as explained in Section 5.1 in detail.

Integer variables are shared (multi-read, multi-write), and may occur in guards and assignments. As a real-time construct, hierarchical timed automata are equipped with *clocks*. Clocks are understood as real-valued variables that change continuous and synchronous as time passes. Clocks can be reset to 0 on transitions, but not set to specific values. Clock values can occur in guards in a syntactically restricted fashion[1]. They can also occur in invariants, but only *downwards closed*, i.e., either as an expression $x < c$ or $x \leq c$, where c is an integer constant.

Hand-shake communication exits between parallel super-states by means of sending (!) or receiving (?) a signal on a *channel*. Two parallel automata can synchronize on transitions by executing them at the same instant. If they are equipped with conflicting variable assignments, the one of the transition labeled with "!" is executed first. Channels may be declared locally, restricting the potential participants in a hand-shake communication. We have to assure, that the control situation remains valid after processing both transitions. A transition t originating in a super-state S cannot synchronize with a transition inside S, for processing t corresponds to rendering S inactive.

There is a top level, where a parallel composition of *fundamental super-states* is specified. They are understood as running in parallel, but *not* put together in an AND super-state for ease of usage: we assume, that system designers will frequently change this part, e.g., to test a controller with respect to different environments put in parallel. Therefore, this parallel composition is realized textually via the $\langle$*system*$\rangle$ tag. For the fundamental super-states, an initial entry has to be declared ($\langle$*globalinit*$\rangle$). Moreover, they are allowed to *terminate* if specified so (`canexit`

[1] To be more precise, clocks x and y may occur in expressions $x - y \backsim c$ and $x \backsim c$, where c is an integer constant and $\backsim \in \{<, >, =, \leq, \geq\}$.

attribute in ⟨*globalinit*⟩), i.e., they can reach a special halt situation that can never be revoked.[2]

In the following, we describe entry and exit of super-states.

Default Entry Optionally, a super-state S can have one entry, that is declared to be the *default* entry. If a transition on the next higher level ends at the border of S, without pointing to an explicit entry, it is assumed to lead to this default entry. In the case that no default entry is declared, such a transition is an error in the model.

History Entry A super-state may be declared to be a history-super-state, by equipping it with a special history entry, designated by a capital H in a circle, Ⓗ. If the super-state is entered via this, the last control location this super-state was in (before it became inactive) is restored. Additionally, all locally declared variables are restored. The locally declared clocks are *not* reset, but kept running. Only clocks explicitly declared as `forgetful clock` are set to 0 on entry via a history entry.

A history entry has to be equipped with a *default history location*, which is entered, if this is the first time the super-state becomes active. This location may be non-basic itself.

Every non-basic sub-state of S of a history super-state H is constrained to have either a history entry or a default-entry. If H is entered via the history entry, and the control points to S, then S is entered either via its history entry, or via the default entry, if S has no history entry.

There is no explicit *deep history entry*, that guarantees to instantiate the history of all enclosed sub-states as well. However, this can be expressed explicitly, by adding a history entry to all such sub-states and their descendants.

Local Clocks Clocks may be declared local to a super-state S. The first time S becomes active, these clocks are set to 0. On re-entry, local clocks are re-set to 0 as well, with one exception: ordinary local clocks are *not* re-set, if S is entered via an history entry. They can be thought of as *kept running* when S becomes inactive. Their value increases in accordance to the global clock. In general, it is not predictable whether it will be re-entered via a history entry or not.

The local clocks declared to be `forgetful clock` are always reset on entry, even this happens via a history entry.

Location Invariants Transitions t to locations carrying an *invariant* can only be taken, if the invariant evaluates to **true** (after possible clock resets executed along t). This generalizes in the situation, where a transition points to a non-basic location: it can only be taken, if the invariants of all reached locations (in case of a fork, there can be several) evaluate to **true** after executing the run-to-completion step.

Forks Forks split the control to parallel sub-states. A fork can carry assignments and clock resets, but no synchronization. Forks may trigger a cascade of other forks, that are all part of the same transition. Forks do not carry guards nor synchronization and they should not have conflicting assignments.

Joins Joins are auxiliary constructs in AND super-states, that move control upward one level, after all sub-states became inactive. A join can carry assignments and clock resets, but no synchronization. Joins may be required to synchronize with other joins, that are all part of the same transition. In our notion, either all or none of them are taken. We do not allow conflicting assignments.

[2] In general, this can violate deadlock freedom. However, it corresponds very much to a situation where a part of the system simply crashes. This aspect is useful, if the model explicitly specifies redundancy.

(Explicit) Exit Explicit exits are denoted by a stub, or—alternatively—by a bullseye (⊙). Pseudo-transitions leading to an exit can only be taken, if the transition step associated with it can be taken as a whole. (This is in conformance with the UML notion of run-to-completion steps.) For notational convenience, various copies of explicit exits can be present in the same super-state. They are identified by sharing the same name.

As a well-formedness constraint, every exit that is reached by a transition, has to be connected to a transition or pseudo-transition on the next-higher level.

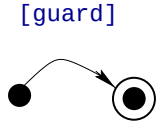


Figure 4.1: Default Exit.

Default Exits The understanding of a default exit is a specially designated exit, that can be reached either *unconditionally* or *guarded* from every enclosed location. This implies, that all non-basic sub-states are required to have default exits as well. If the guard is identical to **true**, this explicitly denotes a super-state to be *interrupt-able*, since it can be left in any case (provided it can synchronize on exit with parallel sub-states; typically, one of them will trigger the interrupt).

From the inside, they are *not* visible in general. But they can be indicated by an unlabeled general exit, see Figure 4.1.
A *configuration* describes a snapshot of the system. In particular, every configuration

1. marks every location of the system as active or inactive
2. denotes one control location for every active XOR super-state
3. defines a value for every global variable and clock, and every local variable and clock of active super-state
4. defines a value for every local variable and local clock for every active super-state and the inactive ones, that contains a history entry

We call a configuration *proper*, if it does not contain pseudo-locations. A *run-to-completion step* is a tuple consisting of a proper source configuration, a step (that is either a proper transition or a sequence containing one proper transition and arbitrary many pseudo-transitions), and a proper target configuration (that is reached from the source configuration via execution of this step).

4.2.2 Dynamics of Transitions

An *execution step* of the model is either an *action step* or a *delay step*. An action step corresponds to executing one run-to-completion step, or—in case of synchronization—two synchronizing run-to-completion steps in an atomic fashion. A run-to-completion step is composed from one proper transition and arbitrary many pseudo transitions. The latter ones can, e.g., encode forks, joins, entries, or exits of sub-states. A run-to-completion step is only enabled, if

1. all the guards in the participating transition parts evaluate to **true**
2. the invariant(s) of the subsequent target location(s) hold after execution of assignments and clock resets

Syntactic restrictions guarantee, that 1. is always equivalent to the case, that the conjunction of these guards are true.

A delay step amounts to incrementing all clock variables by a real number $d > 0$, such that no invariant is violated.

Synchronization on Entry If an AND super-state is entered, every sub-state is entered immediately. This might trigger a cascade of entries, since sub-states are allowed to be AND super-states themselves.

Synchronization on Exit A tree of joins is understood as an indivisible step, i.e., once it is started, it is executed, including the transition following immediately, called *root transition* of

this join. There are no interleavings with other transitions or time delays. If the root transition synchronizes with another transition t, both are taken in parallel.

Pseudo-transitions to exits are allowed to have guards, but no assignment, clock-resets or synchronization labels. This guarantees that, given the conjunction of the guards evaluates to **true** in this configuration, the join can be executed to completion.

Urgency Urgency is a property of transitions and marks them as having priority over delay. If an urgent transition is enabled, the system is not allowed to delay, but must take an action transition as the next step.

Urgency cannot be only be used to resolve conflicts between action transitions and delay transition. An urgent action transition does not have priority over a non-urgent one, if both of them are enabled.

We define different *levels* of urgency as a mean to express priority between urgent transitions. This makes sense only between urgent transitions (urgency $u > 0$). Priority is not mixed with time in our formalism, it is defined on top of the urgency concept.

4.2.3 Lax Input Language

For notational convenience, it makes sense to allow a user to draw state-chart diagrams in a more liberal way. In most cases, this can be safely translated to an explicit formulation. Some examples of this are given in Figure 4.2. Note that arrows on the left-hand side are sometimes replaced by sequences, that contain pseudo-states (stubs), pseudo-transitions, and exactly one ordinary transition. This is the one, where guards, assignments and synchronizations are attached. Following the UML notion of run-to-completion steps, the understanding of the explicit notation is identical with the (usual) interpretation of the lax notation.

In case of ambiguity, we expect a model editor to be clever enough to resolve the choices explicitly. In the following we always assume to have the explicit format, for this makes the task of formalizing the semantics easier.

4.2.4 Differences with UML

We do not yet include the event model of UML. This is work in progress. The channel synchronization is more restricted. The action language on transition is limited to simple computations on integers only. UML defines deep history (recursive) and shallow history (at one hierarchical level), we need only the latter one since deep history can be encoded by the shallow history. UML defines *activities* in states, i.e. processing in a state, we do not have this notion at all[3]. UML defines time events and we model time with clocks.

We define HTA as a core language with limited features. It is expressive enough to encode most of the UML constructs. High level constructs, as used directly in UML state-charts, are not part of HTA and should be seen as user facilities.

[3] this should be removed in the next release of the UML state-charts.

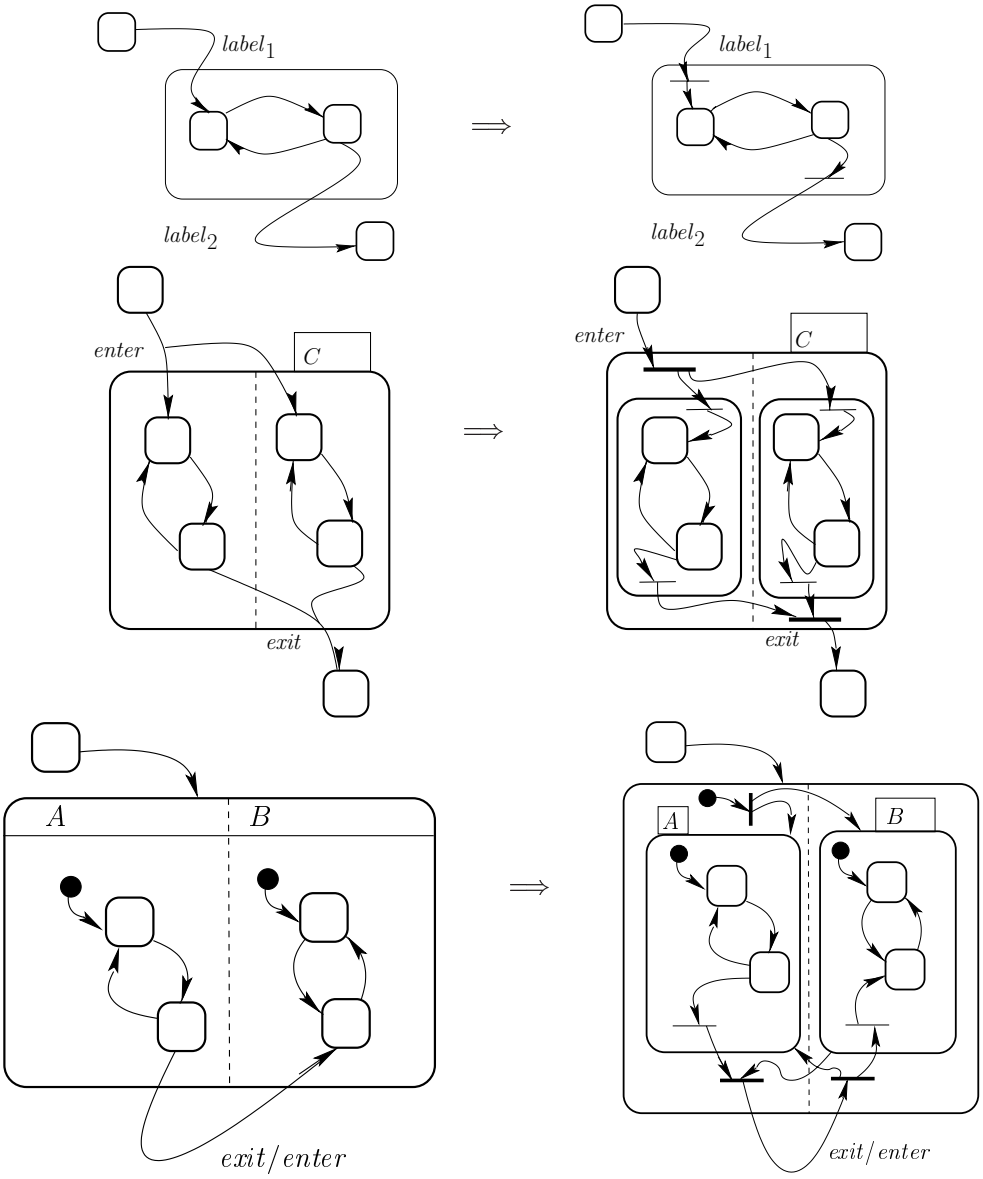


Figure 4.2: Translation of a lax entry formulation to the explicit form.

Chapter 5

Formal Description

5.1 Formal Syntax of HTA

In this section we define the formal syntax of Hierarchical Timed Automata. This is split up in the data parts, the structural parts, and a set of well-formedness constraints.

5.1.1 Data Components

We introduce the data components of Hierarchical Timed Automata, that are used in guards, synchronizations, resets, and assignment expressions. Some of this data is kept local to a generic location, denoted by l.

Integer variables Let V be a finite set of integer variables. $V(l) \subseteq V$ is the set of integer variables local to a super-state l.

Clocks Let $\mathcal{C}$ be a finite set of clock variables. The set $\mathcal{C}(l) \subseteq \mathcal{C}$ denotes the clocks local to a super-state l. If l has a history entry, $\mathcal{C}(l)$ contains only clocks, that are explicitly declared as *forgetful*. Other locally declared clocks of l belong to $\mathcal{C}(root)$.

Channels Let Ch a finite set of synchronization channels. $Ch(l) \subseteq Ch$ is the set of channels that are local to a super-state l, i.e., there cannot be synchronization along a channel $c \in Ch(l)$ between one transition inside l and one outside l.

Synchronizations Ch gives rise to a finite set of channel synchronizations, called $Sync$. For $c \in Ch$, $c?$, $c! \in Sync$. For $s \in Sync$, $\bar{s}$ denotes the matching complementary, i.e., $\bar{c!} = c?$ and $\bar{c?} = c!$.

Guards and invariants A data constraints is a boolean expressions of the form $A \sim A$, where A is an arithmetic expression over V and $\sim \in \{<, >, =, \leq, \geq\}$. A clock constraints is an expressions of the form $x \sim n$ or $x - y \sim n$, where $x, y \in \mathcal{C}$ and $n \in \mathbb{N}$ with $\sim \in \{<, >, =, \leq, \geq\}$. A clock constraint is downward closed, if $\sim \in \{<, =, \leq\}$. A guard is a finite conjunction over data constraints and clock constraints. An invariant is a finite conjunction over downward closed clock constraints. $Guard$ is the set of guards, $Invariant$ is the set of invariants. Both contain additionally the constants **true** and **false**.

Assignments A clock reset is of the form $x := 0$, where $x \in \mathcal{C}$. A data assignment is of the form $v := A$, where $v \in V$ and A an arithmetic expression over V. $Reset$ is the set of clock resets and data assignments.

5.1.2 Structural Components

We give now the formal definition of our Hierarchical Timed Automata.

Definition 3 A hierarchical timed automaton is a tuple $\langle S, S_0, \delta, \sigma, V, \mathcal{C}, Ch, T\rangle$ where

- S is a finite set of locations. $root \in S$ is the root.
- $S_0 \in S$ is a set of initial locations.
- $\delta : S \to 2^S$. δ maps l to all possible sub-states of l. δ is required to give rise to a tree structure with root *root*. We readily extend δ to operate on sets of locations in the obvious way.
- $\sigma : S \to \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ is a type function on locations.
- $V, \mathcal{C}, Ch$ are sets of variables, clocks, and channels. They give rise to *Guard*, *Reset*, *Sync*, and *Invariant* as described in Section 5.1.1.
- $Inv : S \to Invariant$ maps every locations l to an invariant, possibly to the constant **true**.
- $T \subseteq S \times (Guard \times Sync \times Reset \times \mathbb{N}) \times S$ is the set of transitions. A transition connects two locations l and l', has a guard g, an assignment r (including clock resets), and an urgency level u. We use the notation $l \xrightarrow{g,s,r,u} l'$ for this and omit g, s, r, u, when they are necessarily absent (or 0, in the case of u).

Notational conventions We use the predicate notation $TYPE(l)$ for $TYPE \in \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$, $l \in S$. E.g., $AND(l)$ is true, exactly if $\sigma(l) = AND$. The type *HISTORY* is a special case of an entry. We use $HENTRY(l)$ to capture simple entry or history entry, i.e., $HENTRY(l)$ stands for $ENTRY(l) \vee HISTORY(l)$.

We define the parent function

$$\delta^{-1}(l) \stackrel{def}{=} \begin{cases} n, \text{ where } l \in \delta(n) & \text{if } l \neq root \\ \bot & \text{otherwise} \end{cases}$$

We use $\delta^*(l)$ to denote the set of all nested locations of a super-state l, including l. δ^{-*} is the set of all ancestors of l, including l. Moreover we use $\delta^\times(l) \stackrel{def}{=} \delta^*(l) \setminus \{l\}$.
We introduce $\tilde{\delta}$ to refer to the children, that are proper locations.

$$\tilde{\delta}(l) \stackrel{def}{=} \{n \in \delta(l) \mid BASIC(n) \vee XOR(n) \vee AND(n)\}$$

We use $V^+(l)$ to denote the variables in the scope of location l: $V^+(l) = \bigcup_{n \in \delta^{-*}(l)} V(l)$. $\mathcal{C}^+(l)$ and $Ch^+(l)$ are defined analogously.

5.1.3 Well-Formedness Constraints

We give the rules to ensure consistency of a given Hierarchical Timed Automata.

Location constraints We require a number of sanity properties on locations and structure.
The function δ has to give rise to a proper tree rooted at *root*, and $S = \delta^*(root)$.
Basic nodes are empty: $BASIC(l) \Leftrightarrow \delta(l) = \varnothing$.
Sub-states of AND super-state are not basic: $AND(l) \wedge n \in \delta(l) \Rightarrow \neg BASIC(n)$.
Invariants of pseudo-locations are trivial: $HENTRY(l) \vee EXIT(l) \Rightarrow Inv(l) = \mathbf{true}$.

Initial location constraints S_0 has to correspond to a consistent and proper control situation, i.e., $root \in S_0$ and for every $l \in S_0$ it the following holds:

(i) $BASIC(l) \vee XOR(l) \vee AND(l)$,
(ii) $l = root \vee \delta^{-1}(l) \in S_0$,
(iii) $XOR(l) \Rightarrow |\delta(l) \cap S_0| = 1$, and
(iv) $AND(l) \Rightarrow \delta(l) \cap S_0 = \tilde{\delta}(l)$.

Variable constraints We explicitly disallow conflict in assignments in synchronizing transitions: It holds that $l_1 \xrightarrow{g,c!,r,u} l_1'$, $l_2 \xrightarrow{g',c?,r',u'} l_2' \in T \Rightarrow vars(r) \cap vars(r') = \varnothing$, where $vars(r)$ is the set of integer variables occurring in r. We require an analogous constraint to hold for the pseudo-transitions originating in the entry of an AND super-state.

Static scope: For $l \xrightarrow{g,s,r,u} l' \in T$, g, r are defined over $V^+(\delta^{-1}(l)) \cup \mathcal{C}^+(\delta^{-1}(l))$ and s is defined over $Ch^+(\delta^{-1}(l))$.

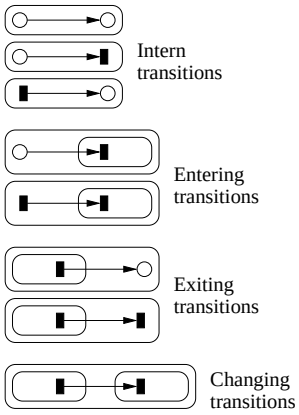
Entry constraints Let $e \in S$, $HENTRY(e)$. If $\text{XOR}(\delta^{-1}(l))$, then T contains exactly one transition $e \xrightarrow{r} l'$. If $\text{AND}(\delta^{-1}(l))$, then T contains exactly one transitions $e \xrightarrow{r} e_i$ for every proper sub-state $l_i \in \tilde{\delta}(\delta^{-1}(l))$, and $e_i \in \delta(l_i)$.

In case of $HISTORY(e)$, outgoing transitions declare the default history locations.

If a super-state s has a history entry, then every sub-state l of s has to provide either a history entry or a default entry.

Transition constraints Transitions have to respect the structure given in δ and cannot cross levels in the hierarchy, except via connecting to entries or exits. The set of legal transitions is given in Table 5.1 Note that transitions cannot lead directly from entries to exits.

Transitions $l \xrightarrow{g,s,r,u} l'$ with $HENTRY(l)$ or $EXIT(l')$ are called *pseudo-transitions*. They are restricted in the sense, that they cannot carry synchronizations or urgency flags, and only either guards or assignments. For $HENTRY(l)$, only pseudo-transition of the form $l \xrightarrow{r} l'$ are allowed. For $EXIT(l')$, only pseudo-transition of the form $l \xrightarrow{g} l'$ are allowed. For $EXIT(l) \wedge EXIT(l')$, this is further restricted to be of the form $l \to l'$.



Comment	l	l'	Constraint
Intern	$BASIC$	$BASIC$	$\delta^{-1}(l) = \delta^{-1}(l')$
	$BASIC$	$EXIT$	
	$HENTRY$	$BASIC$	
Entering and fork	$BASIC$	$HENTRY$	$\delta^{-1}(l) = \delta^{-2}(l')$
	$HENTRY$	$HENTRY$	
Exiting and join	$EXIT$	$BASIC(l)$	$\delta^{-2}(l) = \delta^{-1}(l')$
	$EXIT$	$EXIT$	
Changing	$EXIT$	$HENTRY$	$\delta^{-2}(l) = \delta^{-2}(l')$

Table 5.1: Overview over all legal transitions $l \xrightarrow{g,s,r,u} l'$.

5.2 Operational Semantics of HTA

We present the operational semantics of our Hierarchical Timed Automata model. A configuration captures a snapshot of the system, i.e., the active locations, the integer variable values, the clock values, and the history of some super-states. Configurations are of the form (ρ, μ, ν, θ), where

- $\rho : S \to 2^S$ captures the control situation. ρ can be understood as a partial, dynamic version of δ, that maps every super-state s to the set of active sub-states. If a super-state s is not active, $\rho(s) = \varnothing$. We define $Active(l) \stackrel{def}{=} l \in \rho^\times(root)$, where $\rho^\times(l)$ is the set of all active sub-states of l. Notice that $Active(l) \Leftrightarrow l \in \rho(\delta^{-1}(l))$.
- $\mu : S \to (\mathbb{Z})^*$. μ gives the valuation of the local integer variables of a super-state l as a finite tuple of integer numbers. If $\neg Active(l)$ then $\mu(l) = \lambda$ (the empty tuple). If $Active(l)$ then we require that $|\mu(l)| = |V(l)|$ and μ is consistent with respect to the value of shared variables

(i.e., always maps to the same value). We use $\mu(l)(a)$ to denote the value of $a \in V(l)$. When entering a non-basic location, local variables are added to μ and set to an initial value (0 by default). We use the shorthand $0^{V(l)}$ for the tuple $(0, 0 \ldots 0)$ with arity $|V(l)|$.

- $\nu : S \to (\mathbb{R}^+)^*$. ν gives the real valuation of the clocks $\mathcal{C}(l)$ visible at location l, thus $|\nu(l)| = |\mathcal{C}(l)|$. If $\neg \mathit{Active}(l)$ then $\nu(l) = \lambda$.
- θ reflects the history, that might be restored by entering super-states via history entries. It is split up in the two functions θ_{state} and θ_{var}, where $\theta_{state}(l)$ returns the last visited sub-state of l—or an entry of the sub-state, in the case where the sub-state is not basic—(to restore $\rho(l)$), and $\theta_{var}(l)$ returns a vector of values for the local integer variables.
  There is no history for clocks at the semantics level, all non-forgetful clocks belong to $\mathcal{C}(\mathit{root})$.

History We capture the existence of a history entry with the predicate $\mathit{HasHistory}(l) \stackrel{def}{=} \exists n \in \delta(l).\ \mathit{HISTORY}(n)$. If $\mathit{HasHistory}(l)$ holds, the term $\mathit{HEntry}(l)$ denotes the unique history entry of l. If $\mathit{HasHistory}(l)$ does not holds, the term $\mathit{HEntry}(l)$ denotes the default entry of l. If l is basic $\mathit{HEntry}(l) = l$. If none of the above is the case, then $\mathit{HEntry}(l)$ is undefined.

Initially, $\forall l \in S.\mathit{HasHistory}(l) \Rightarrow \theta_{state}(l) = \mathit{HEntry}(l) \wedge \theta_{var}(l) = 0^{V(l)}$.

Reached locations by forks In order to denote the set of locations reached by following a fork, we define the function $\mathit{Targets}_\theta : 2^S \to 2^S$ relative to θ.

$$\mathit{Targets}_\theta(L) \stackrel{def}{=} L \cup \bigcup_{l \in L} \{n \mid n \in \theta_{state}(l) \ \wedge \ \mathit{HISTORY}(l)\} \cup \{n \mid l \xrightarrow{r} n \ \wedge \ \mathit{ENTRY}(l)\}$$

We use the notation $\mathit{Targets}_\theta(l)$ for $\mathit{Targets}_\theta(\{l\})$, if the argument is a singleton. $\mathit{Targets}^*_\theta$ is the reflexive transitive closure of $\mathit{Targets}_\theta$.

Configuration vector transformation Taking a transition $t : l \xrightarrow{g,s,r,u} l'$ entails in general 1. executing a join to exit l, 2. taking the proper transition t itself, and 3. executing a fork at l'. If l (respectively l') is a basic location, part 1. (respectively 3.) is trivial. Together, this defines a run-to-completion step. We represent a run-to-completion step formally by a transformation function $\mathcal{T}_t$, which depends on a particular transition t. The three parts of this step are described as follows.

1. *join:*
   (ρ, μ, ν, θ) is transformed to $(\rho^1, \mu^1, \nu^1, \theta^1)$ as follows:
   ρ is updated to $\rho^1 := \rho[\forall n \in \rho^\times(l).\ n \mapsto \varnothing]$.
   μ is updated to $\mu^1 := \mu[\forall n \in \rho^\times(l).\ n \mapsto \lambda]$.
   ν is updated to $\nu^1 := \nu[\forall n \in \rho^\times(l).\ n \mapsto \lambda]$.

   If $\mathit{EXIT}(l)$, the history is recorded. Let H be the set of super-states $h \in \rho^\times(\delta^{-1}(l))$, where $\mathit{HasHistory}(h)$ holds. Then
   $$\theta^1_{state} := \theta_{state}[\forall h \in H.\ h \mapsto \mathit{HEntry}(\rho(h))] \quad \text{and}$$
   $$\theta^1_{var} := \theta_{var}[\forall h \in H.\ h \mapsto \mu(h)].$$
   If $\neg \mathit{EXIT}(l)$ or $H = \varnothing$, then $\theta^1 := \theta$.
2. *proper transition part:*
   $(\rho^1, \mu^1, \nu^1, \theta^1)$ is transformed to $(\rho^2, \mu^2, \nu^2, \theta^2) := (\rho^1[l'/l], r(\mu^1), r(\nu^1), \theta^1)$. $r(\mu^1)$ denotes the updated values of the integers after the assignments and $r(\nu^1)$ the updated clocks after the resets.
3. *fork:*
   $(\rho^2, \mu^2, \nu^2, \theta^2)$ is transformed to $(\rho^3, \mu^3, \nu^3, \theta^3)$ by moving the control to all proper locations reached by the fork, i.e., those in $\mathit{Targets}^*_{\theta^2}(l')$. Note that $\rho^2(n) = \varnothing$ for all $n \in \delta^\times(l')$. Thus we can compute ρ^3 as follows:

$\rho^3 := \rho^2$
FORALL $n \in \mathit{Targets}^*_{\theta^2}(l')$
IF $\mathit{ENTRY}(n)$
THEN $\rho^3(\delta^{-2}(n)) := \rho^3(\delta^{-2}(n)) \cup \{\delta^{-1}(n)\}$
ELSE $\rho^3(\delta^{-1}(n)) := \{n\}$ /⋆ *BASIC* ⋆/

μ^3 is derived from μ^2 by first initializing all local variables of the super-states s in $\mathit{Targets}^*_{\theta^2}(l')$, i.e., $\mu^3(V(s)) := 0^{V(s)}$. If $\mathit{HasHistory}(s)$, $\theta_{var}(s)$ is used instead of $0^{V(s)}$. Then all variable assignments and clock-resets along the pseudo-transitions belonging to this fork are executed to update μ^3 and ν^3. The history does not change, θ^3 is identical to θ^2.

Note that parts 1. and 3. correspond to the identity transformation, if l and l' are basic locations.

We define the configuration vector transformation $\mathcal{T}_t$ for a transition $t : l \xrightarrow{g,s,r,u} l'$:

$$\mathcal{T}_t(\rho,\mu,\nu,\theta) \stackrel{def}{=} (\rho^3,\mu^3,\nu^3,\theta^3)$$

If the context is unambiguous, we use $\rho^{\mathcal{T}_t}$ and $\nu^{\mathcal{T}_t}$ for the parts ρ^3 respectively ν^3 of the transformed configuration corresponding to transition t.

Starting points for joins A super-state s can only be exited, if all its parallel sub-states can synchronize on this exit. For an exit $l \in \delta(s)$ we recursively define the family of sets of exits $\mathit{PreExitSets}(l)$. Each element X of $\mathit{PreExitSets}(l)$ is itself a set of exits. If transitions are enabled to all exits in X, then all sub-states can synchronize.

$$\mathit{PreExitSets}(l) \stackrel{def}{=} \begin{cases} \bigcup_{n_1,\ldots,n_k} \boxtimes_{1\le i\le k} \mathit{PreExitSets}(n_i), \text{ where} \\ \quad k = |\tilde{\delta}(\delta^{-1}(l))|,\ \{n_1,\ldots,n_k\} \subseteq \delta^{\times}(\delta^{-1}(l)), \\ \quad \forall i.\, \mathit{EXIT}(n_i),\ \{\delta^{-1}(n_1),\ldots,\delta^{-1}(n_k)\} = \tilde{\delta}(l) \end{cases} \text{if } \begin{array}{l}\mathit{EXIT}(l)\wedge \\ \mathit{AND}(\delta^{-1}(l))\end{array} \\ \\ \begin{cases} \bigcup_{m\in\delta(\delta^{-1}(l))} \mathit{PreExitSets}(m), \text{ where } m \xrightarrow{g,r} l \in T \\ \quad \cup\ \{\{l\}\} \end{cases} \text{if } \begin{array}{l}\mathit{EXIT}(l)\wedge \\ \mathit{XOR}(\delta^{-1}(l))\end{array} \\ \\ \{\{\}\} \qquad \text{if } \mathit{BASIC}(l) \end{cases}$$

Here, the operator $\boxtimes : (2^{2^S}) \times (2^{2^S}) \to 2^{2^S}$ is a product over families of sets, i.e., it maps $(\{A_1,\ldots,A_a\},\{B_1,\ldots,B_b\})$ to $\{A_1\cup B_1, A_1\cup B_2,\ldots,A_a\cup B_b\}$ and is extended to operate on an arbitrary finite number of arguments in the obvious way.

Rule predicates To give the rules, we need to define predicates that evaluate conditions on the dynamic tree ρ. We introduce the set set of active leaves (in the tree described by ρ), which are the innermost active states in a super-state l:

$$\mathit{Leaves}(\rho,l) \stackrel{def}{=} \{n \in \rho^{\times}(l) \mid \rho(n) = \varnothing\}$$

The predicate expressing that all the sub-states of a state l can synchronize on a join is:

$$\begin{array}{rl} \mathit{JoinEnabled}(\rho,\mu,\nu,l) \stackrel{def}{=} & \mathit{BASIC}(l) \vee \\ & \exists X \in \mathit{PreExitSets}(l).\ \forall n \in \mathit{Leaves}(\rho,l).\ \exists n' \in X.\ n \xrightarrow{g} n' \ \wedge\ g(\mu,\nu) \end{array}$$

Note that *JoinEnabled* is trivially true for a basic location l.

For the invariants of a location we use a function $\mathit{Inv}_\nu : S \to \{\mathbf{true},\mathbf{false}\}$, that evaluates the invariant of a given location with respect to a clock evaluation ν. We use the predicate $\mathit{Inv}(\rho,\nu)$ to express, that for control situation ρ and clock valuation ν all invariants are satisfied.

$$\mathit{Inv}(\rho,\nu) \stackrel{def}{=} \bigwedge_{n\in\rho^{\times}(root)} \mathit{Inv}_\nu(n)$$

We introduce the predicate *TransitionEnabled* over transitions $t : l \xrightarrow{g,s,r,u} l'$, that evaluates to **true**, if t is enabled.

$$\begin{aligned} &\mathit{TransitionEnabled}(t : l \xrightarrow{g,s,r,u} l', \rho, \mu, \nu) \stackrel{def}{=} \\ &\qquad g(\mu,\nu) \wedge \mathit{JoinEnabled}(\rho,\mu,\nu,l) \wedge \mathit{Inv}(\rho^{\mathcal{T}_t}, \nu^{\mathcal{T}_t}) \wedge \neg \mathit{EXIT}(l') \end{aligned}$$

Since urgency has precedence over delay, we have to capture the global situation, where some urgent transition is enabled. We do this via the predicate *UrgentEnabled* over a configuration.

$$\begin{aligned} \mathit{UrgentEnabled}(\rho,\mu,\nu) \quad \stackrel{def}{=} \quad & \exists t : l \xrightarrow{g,r,u} l'.\ \mathit{TransitionEnabled}(t,\rho,\mu,\nu) \ \wedge\ u \\ \vee \quad & \exists t_1 : l_1 \xrightarrow{g_1,s,r_1,u_1} l_1', t_2 : l_2 \xrightarrow{g_2,\bar{s},r_2,u_2} l_2'. \\ & \qquad \mathit{TransitionEnabled}(t_1,\rho,\mu,\nu) \wedge \\ & \qquad \mathit{TransitionEnabled}(t_2,\rho,\mu,\nu) \ \wedge\ (u_1 \vee u_2) \end{aligned}$$

When several urgent transitions are enabled and are in conflict, then the most urgent one is taken. We compute this urgency level with the function *EnabledUrgency* over a configuration.

$$\mathit{EnabledUrgency}(\rho,\mu,\nu,t) \stackrel{def}{=} max_{0,u(t') \mid t' \in \mathit{Conflict}(\rho,\mu,\nu,t)} u$$

where $u(t')$ is the urgency of a transition t' belonging to the set of transitions in conflict with t. A transition $t' : l_1' \xrightarrow{g',s',r',u'} l_2'$ is in conflict with a transition $t : l_1 \xrightarrow{g,s,r,u} l_2$ iff

- s' is empty and $l_1' \in^* (l_1) \cup \delta^{-*}(l_1)$, $\mathit{TransitionEnabled}(t',\rho,\mu,\nu)$
- or s' is not empty and $l_1' \in^* (l_1) \cup \delta^{-*}(l_1)$, $\mathit{TransitionEnabled}(t',\rho,\mu,\nu)$, $\exists t'' : l_1'' \xrightarrow{g'',\bar{s},r'',u''} l_2''$. $\mathit{TransitionEnabled}(t'',\rho,\mu,\nu)$

Rules We give now the action rule. It is not possible to break it in join, action, and fork because the join can be taken only if the action is enabled and the action is taken only if the invariants still hold after the fork.

$$\frac{\mathit{TransitionEnabled}(t : l \xrightarrow{g,r,u} l',\ \rho,\mu,\nu) \quad u = 0 \vee u \geq \mathit{EnabledUrgency}(\rho,\mu,\nu,t)}{(\rho,\mu,\nu,\theta) \xrightarrow{t} \mathcal{T}_t(\rho,\mu,\nu,\theta)}\ \mathit{action}$$

Here g is the guard of the transition and r the set of resets and assignments. The urgency is used as a priority between urgent enabled transitions. Non urgent transitions are not compared with urgent ones. This rule applies for action transitions between basic locations as well as super-states. In the later case, this includes the appropriate joins and/or fork operations.

The delay transition rule is:

$$\frac{\mathit{Inv}(l)(\rho,\nu+d) \qquad \neg\, \mathit{UrgentEnabled}(\rho,\mu,\nu)}{(\rho,\mu,\nu,\theta) \xrightarrow{d} (\rho,\mu,\nu+d,\theta)}\ \mathit{delay}$$

where $\nu + d$ stands for the current clock assignment plus the delay for all the clocks. Time elapses in a configuration only when all invariants are satisfied and there is no urgent transition enabled.

The last transition rule reflects the situation, where two action transitions synchronize via a channel c.

$$\frac{\begin{array}{l}\mathit{TransitionEnabled}(t_1 : l_1 \xrightarrow{g_1,c!,r_1,u_1} l_1',\ \rho,\mu,\nu) \quad l_1 \notin \delta^{\times}(l_2) \quad u_1 = 0 \vee u_1 \geq \mathit{EnabledUrgency}(\rho,\mu,\nu,t_1) \\ \mathit{TransitionEnabled}(t_2 : l_2 \xrightarrow{g_2,c?,r_2,u_2} l_2',\ \rho,\mu,\nu) \quad l_2 \notin \delta^{\times}(l_1) \quad u_2 = 0 \vee u_2 \geq \mathit{EnabledUrgency}(\rho,\mu,\nu,t_2)\end{array}}{(\rho,\mu,\nu,\theta) \xrightarrow{t_1,t_2} \mathcal{T}_{t_2} \circ \mathcal{T}_{t_1}(\rho,\mu,\nu,\theta)}\ \mathit{sync}$$

We choose a particular order here but it is not crucial since our well-formedness constraints ensure, that the assignments cannot conflict with each other.

If no action transition is enabled or becomes enabled when time progresses, we have a *deadlock* configuration, which is typically a bad thing. If in addition time is prevented to elapse, this is a *time stopping deadlock*. Usually this is an error in the model, since it does not correspond to any real world behavior.

Our rules describe all legal sequences of transitions. A *trace* is a finite or infinite sequence of legal transitions, that start at the initial configuration S_0, with all variables and clocks set to 0. For our purposes it suffices to associate a Hierarchical Timed Automata semantically with the (typically infinite) set of all derivable traces.

Chapter 6

Translation to UPPAAL

In this Section we give a detailed description of our flattening procedure, that translates our Hierarchical Timed Automata model to to a parallel composition of (flat) UPPAAL timed automata [LPY97]. For both models we have a syntactic representation via a XML document type definition. The translator written in Java takes as input a Hierarchical Timed Automata model respecting the syntax of the document type definition (DTD). It outputs a timed automata model following the syntax of the DTD. The timed automata model is then fed to the UPPAAL verification tool from version 3.2 on. The code was written by Oliver Möller and it can be found at `http://www.brics.dk/~omoeller/hta/vanilla-1/`.

The fundamental concept of our flattening algorithm is the translation of every hierarchical super-state into one UPPAAL automaton. All these automatons are put in parallel. This can lead to an exponential blow-up in terms of templates, or in other words, of the model size. This is a consequence of the fact that hierarchical models can be exponentially more concise [AKY99]. Some auxiliary structures have to be introduced in order to mimic the behavior of hierarchical machines adequately.

6.1 UPPAAL Timed Automata

UPPAAL [LPY97] is a tool box for modeling, verification and simulation or real-time systems developed jointly by Uppsala University and Aalborg University. It is appropriate for systems that can be described as collection of non-deterministic parallel processes. The model used in UPPAAL is the timed automaton and corresponds to the flat version of our Hierarchical Timed Automata where each process is described as a state machine with finite control structure, real-valued clocks and integers. Processes communicate through channels and (or) shared variables [KGLY95]. The tool has been successfully applied in many case studies [LPY98, LP97, HSLL97].

UPPAAL features committed locations, as an extension to timed automata. This special modeling construct is indicated by a `c` on the locations. If a committed location l is active, it must be left as soon as possible, i.e., no time delay is possible and all transitions originating in non-committed locations are blocked, unless they synchronize with a transition leaving l. Committed locations can be used to encode more complex behavior, but also to reduce the number of possible interleavings and thus render state space exploration more efficient. The translation makes heavy use of these committed locations to serialize forks and joins that are intrinsically multi-synchronizations.

As another extension of timed automata, UPPAAL supports integer variables. These variables can be declared local to an automaton for a private use or global for sharing between different automata. In the current translation variables are renamed and declared globally.

6.2 Translation Algorithm

The basic concept of the procedure is the translation of instantiated templates. For every super-state occurring in the HTA model, one UPPAAL template is constructed. However, this cannot be done in an transducer fashion. Since parallel states synchronize on exit, information about exits depends on other parts, that may not have been translated yet.

Thus the translation has three phases: collection of instantiations, computation of global joins, and post-processing channel communication.

For sake of clarity, we choose to omit various thinkable optimizations. For example, XOR sub-states of XOR super-states or AND sub-states of AND super-states are not lifted, even if there are no local variables on the lower levels.

We present the pacemaker case study, a well known example used in UML textbooks [Dou99].

6.2.1 Phase I: Collection of Instantiations

In this phase, the (implicit) hierarchical instantiation tree is traversed and for every hierarchical super-state, the skeleton of a (flat) template is constructed.

Initially, the direct children of the root are on the stack, i.e., the fundamental super-states as contained in the ⟨*system*⟩ element. The algorithm *instantiateTemplates* is given in appendix 11 How exactly the super-states I are translated is dependent on their type, that is either XOR or AND.

XOR:
- have basic locations and transitions
- may contain super-states (⟨*component*⟩s)
- have at least one ⟨*entry*⟩
- may have ⟨*exit*⟩s
- entries are connected to locations or entries of sub-states
- exits are reached from locations or exits of sub-states

AND:
- have no basic locations, no transitions
- have at least two ⟨*component*⟩s
- have at least one ⟨*entry*⟩
- entries correspond to ⟨*fork*⟩s
- may have ⟨*exit*⟩s
- exits correspond to ⟨*join*⟩s

Translation of XOR Super-states. In a hierarchical *XOR* template X, at most one location is active at the same point in time. To represent the situation that none is active, we introduce —in the translation $\hat{X}$—the special location `X_IDLE`, which is also the initial state. All entries are translated by a transition from `X_IDLE`. For every sub-state S of X we introduce a location `S_ACTIVE_IN_X` in $\hat{X}$.

Moreover, for every entry e of S we introduce an auxiliary location in $\hat{X}$, called `X_AUX_S_`e. These are declared committed and are connected to `S_ACTIVE_IN_X` with a transition, that synchronizes on a signal `enter_S_in_X_via_`e. Transitions leading originally to a S-entry e in X are represented in the translation by leading to `X_AUX_S_`e and trigger—without interleaving with other components—the activation of the sub-state S.

Exits of this sub-state S are more complicated, for they are only possible, if all non-basic sub-states of S can exit. This is described as global joins, see Section 6.2.2.

If super-state X is inactivated, this is realized in the translation $\hat{X}$ by transitions to `IDLE_X`, that are triggered by an `exit_X` synchronization channel. If the super-state X has a default exit, every non-auxiliary location in $\hat{X}$ has a transition to `IDLE_X`.

Translation of AND Super-states. A hierarchical *AND* machine A is a parallel composition of sub-machines, where either none or all of them are active. In the translation $\hat{A}$ (Figure 6.1), these situations are represented by the locations `A_IDLE` and `A_ACTIVE`. If A is activated, this

is always specific to a designated entry $\mathtt{e}_i$ of A. The sub-machines S_i of A are all entered, but the signals `enter_Si_via_ej` depend on the choice of `ej`. Therefore, for every entry there is a separate chain leading from `A_IDLE` to `A_ACTIVE`. The auxiliary locations in between are declared committed (marked by a `c`), thus there are no time delays possible.

The exit of A is represented in $\hat{A}$ via a transition from `A_ACTIVE` to `A_IDLE`, which carries the synchronization signal `exit_A`.

Section 6.2.5 illustrates this translation with a concrete small example.

6.2.2 Phase II: Computation of Global Joins

Transitions originating from super-states are a subtle issue, for they may require a cascade of sub-state exits—called *global join*—in order to be taken. The global join can be seen as a tree with the leaves nested at arbitrary sub-state level. The translation serializes the multi-synchronization of the join.

In Figure 6.2 (a), the sub-states `S1`, `S2`, and `S3` have to be exited, before `LOC` can be reached. If `Sn` is active in `S2`, it has to be exited as well. In phase I of our flattening algorithm, the output GJ collects the topmost components, that have to be exited, if a transition (like to `LOC` in Figure 6.2) has to be translated. One entry in GJ can give rise to a number of global joins, possibly exponential in the depth of hierarchical structure. In Figure 6.2, the locations `L3a` and `L3b` can be treated uniformly, but the location `L1` has to be encoded in a *different* global join, where there is no exit of sub-state `Sn`.

Every possible global join is translated to a sequence like in Figure 6.2 (b). The auxiliary variable `trigger` keeps track of the number of active basic locations, that are connected to this global join via a transition to an exit. It has to reach the threshold value `N` to enable the first transition. Moreover, it has to be possible to mimic the transition to `LOC`, i.e., the `guard` (if any) has to be satisfied and synchronization (if any) has to be possible. Synchronization is not possible with transitions *inside* S1. If this situation arises in the given HTA model, we introduce new channels to avoid this conflict and duplicate transitions accordingly, see 6.2.3.

The algorithm *expandGlogalJoins* is given in appendix 11

6.2.3 Phase III: Post-processing Channel Communication

If a transition in the Hierarchical Timed Automata formalism starts at a non-basic state S and carries a synchronization, it cannot synchronize with a transition *inside* S. Since the sub-state/super-state relation is lost in the translation, we resolve this scope conflict explicitly. We do this by introducing duplications of channels and transitions.

We start with a priority queue Q over transitions that possibly can cause a conflict. These elements were collected during the construction of the global joins. Q is sorted obeying the partial order introduced by the sub-state/super-state relation on instantiations. The post-processing algorithm *postprocessChannels* is given in appendix 11.

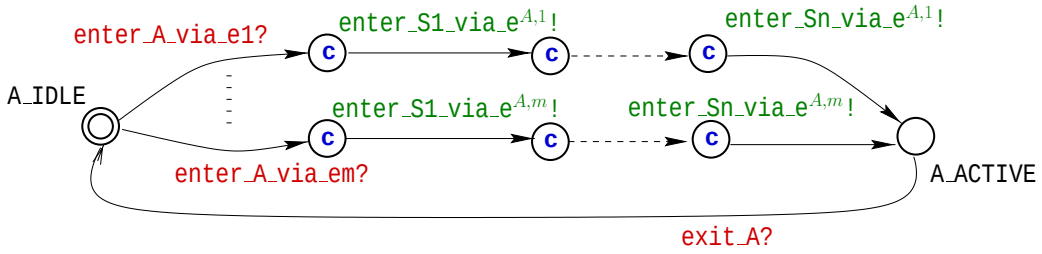
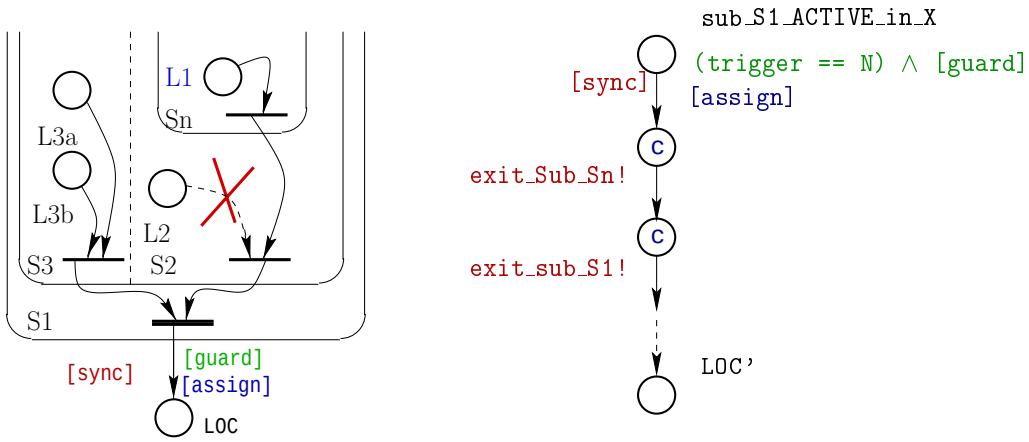


Figure 6.1: Translation of entering and exiting an *AND* component.



(a) Part of Hierarchical Timed Automata X *(b) Translation a the global join in $\hat{X}$*

Figure 6.2: The exit of `S1` in super-state X gives rise to a number of global joins.

6.2.4 Correctness of the Translation

Starting at the root level, we can define a correspondence between every legal global state of the HTA model and its translation into UPPAAL timed automata.

Every super-state S in the Hierarchical Timed Automata model corresponds exactly to one UPPAAL timed automaton $\hat{S}$. For proper configurations, we can relate ρ in the Hierarchical Timed Automata model to a *control vector* $\hat{\rho}$ in the UPPAAL model. Proper configurations for Hierarchical Timed Automata do not include pseudo-states, and for UPPAAL models they do not include committed locations. Sequences of committed locations in UPPAAL are taken completely or not at all, as for the run-to-completion step on forks and joins.

For an UPPAAL automaton U, $\hat{\rho}(U)$ denotes the active location of U. For all XOR super-states X, $\hat{\rho}$ contains at position $\hat{X}$ either a translation of a basic state $\hat{l}$, `sub_S_active_in_X`, or `IDLE`, depending on whether $\rho(X)$ maps to a basic state, to a sub-state S, or to $\varnothing$. For *AND* super-state A, $\hat{\rho}(\hat{A}) =$ `IDLE` if $\rho(A) = \perp$ and $\hat{\rho}(\hat{A}) = \{\hat{S} \mid S \text{ parallel sub-state of } A\}$ otherwise. The value of the introduced auxiliary variables is completely determined by the current control location, i.e., it is redundant for the configuration and only serves to enable or disable transitions.

Proposition A hierarchical state $s = (\rho, u)$ is reachable if and only if a corresponding state $\hat{s} = (\hat{\rho}, u)$ is reachable.

Since entries and exits in the UPPAAL translation are guaranteed to take place without time delay (due to encoding with committed locations), data and clock evaluations u carries over without changes. If a hierarchical trace t exits, it can be mimicked by the translation in each step. Likewise, if a translation $\hat{t}$ of a hierarchical trace is legal in the UPPAAL model, this is due to a sound sequence of entries and exits and corresponds to a trace in the Hierarchical Timed Automata formalism.

6.2.5 Example

We give a short example to illustrate the translation. Figure 6.3 shows a part of a HTA taken from a fictious larger model. The AND state `A` is entered by a default entry or by an explicit fork. These entries lead to two different configurations. The exit of `A` is done by an explicit join or the default exit. The AND state `A` has the invariant $x < 5$ and the sub-state `L1` the invariant $y < 5$.

The translation of this HTA is given in figure 6.4. The main automaton has the two different ways of entering `A` and the two possible global joins of this model. The translated automaton `A`

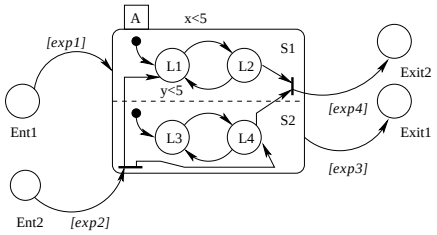


Figure 6.3: Original HTA.

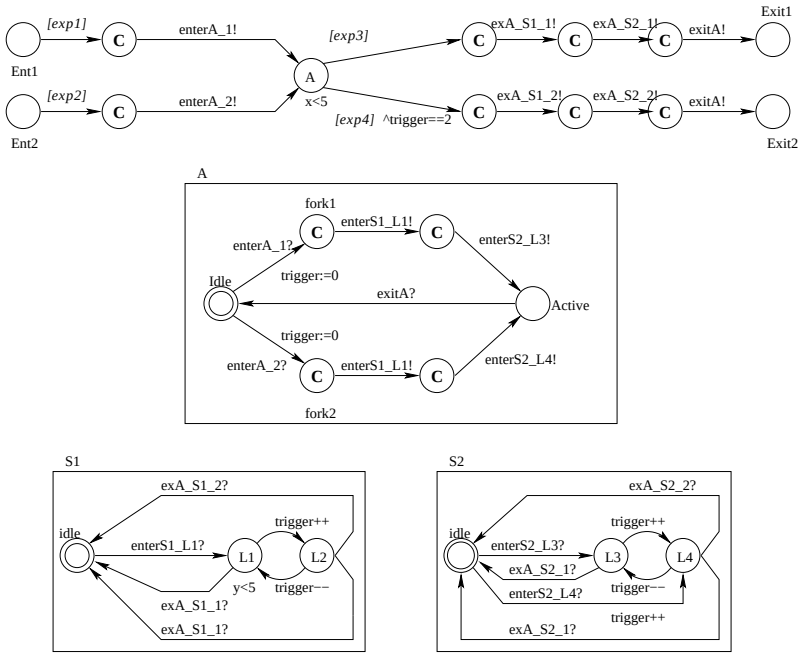


Figure 6.4: Translated HTA.

performs the forks and waits in the state `Active` until (the original) `A` is left. The UPPAAL init states are represented as double circles. The two sub-states `S1` and `S2` are entered from their idle state and they return to this idle state upon exit. The default exit starts from all the sub-states. The explicit join is constrained by the `trigger` variable.

6.3 The Pacemaker Case Study

In this section we apply our flattening procedure on a Hierarchical Timed Automata version of a cardiac pacemaker model. This model is strongly motivated by the often-used UML design example, see e.g. [Dou99]. The pacemaker is put in parallel with a model of a human heart and a programmer, who changes operation modes on the pacemaker. We translate the Hierarchical Timed Automata model of this composition to an equivalent (flat) UPPAAL timed automata model and explain the obtained automata in detail. Additionally, we report on run-time data of the formal verification of this translation with respect to safety and response properties.

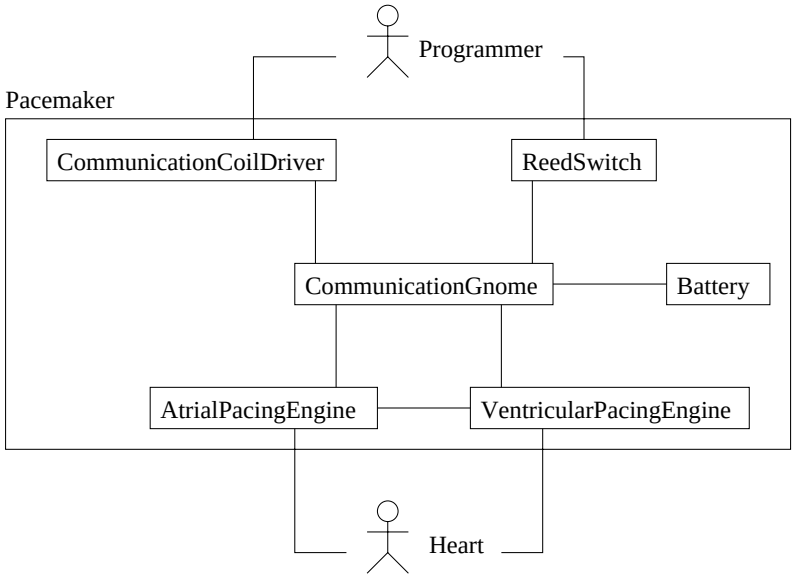


Figure 6.5: Object model of the pacemaker.

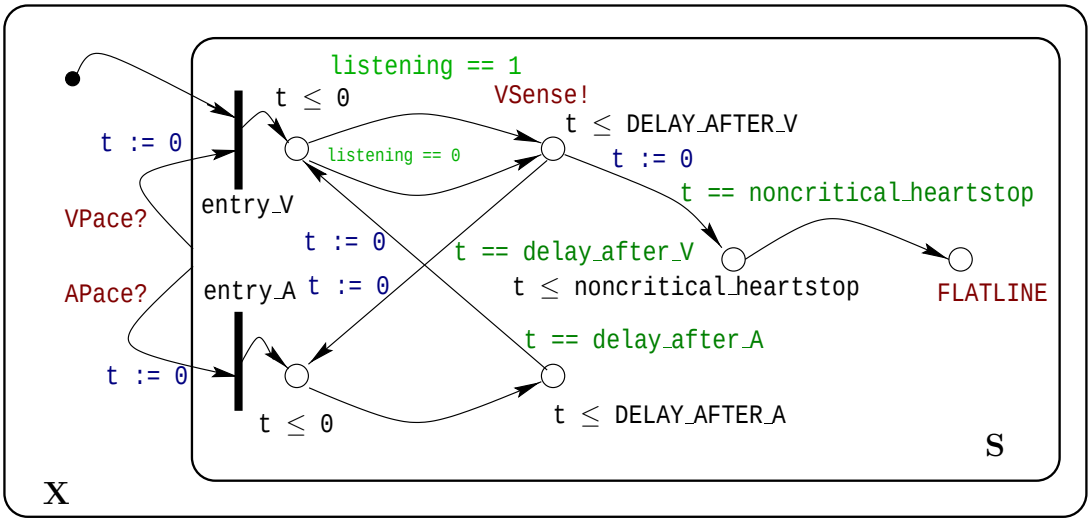


Figure 6.6: The simplified model of the human heart.

Model

Figure 6.5 shows the object model of the pacemaker. All the relations have cardinality 1-1. The programmer is an external actor to the system, he represents the physician who configures and controls the pacemaker. The CommunicationCoilDriver handles communication via the telemetry coil. The ReedSwitch is a security switch to prevent inadvertent reprogramming of the pacemaker. The two pacing engines are in charge of pacing a particular heart chamber (atrium and ventricular). The heart is an actor providing cardiac electrical status to the pacemaker and receiving electrical stimuli from it.

Pacing modes refers to the different types of pacing delivered. As a naming convention, pacing modes are encoding with a three letter abbreviation: the first letter is the paced heart chamber, A or V. The second letter is the sensed chamber, A or V. The third letter is the type of pacing, I

for inhibited or T for triggered. In the inhibited mode, the specified chamber is sensed and if no heartbeat occurs within a specified amount of time, then the chamber is paced. In the triggered mode the chamber is always paced.

In our model we consider only the operation modes idle, VVT, VVI and AVI. The ventricular chamber is sensed and the atrial and the ventricular chamber are paced. The simplified heart model takes into account only the (left) atrial and ventricular chambers. A healthy heart contracts these chambers in steady rhythm, dictated by the time delays `DELAY_AFTER_V` and `DELAY_AFTER_A`. The simplified hear model is depicted in figure 6.6. The local clock `t` is used to model the rhythm and the heart beat is sensed with the `Vsense` synchronization channel. The heart may stop to beat, but it should not do so too long, otherwise the critical state `FLATLINE` is reached. The pacemaker sends signals to the atrial and ventricular chambers, synchronizing with the `APace` or the `VPace` channels.

The reader is referred to [DM01] for a more complete description description of the model.

6.3.1 Translation to UPPAAL

	HTA model	UPPAAL model
# proper control locations	35	45
# pseudo-states / committed locations	33	63
# transitions	47	177
# variables and constants	33	72
# clocks	6	6

Table 6.1: Comparison of the Hierarchical Timed Automata and the flat UPPAAL models.

The HTA model consists of the programmer, the heart, and the pacemaker running in parallel. The translation gives 14 automata corresponding to the different hierarchies as described in the section 6. Table 6.1 compares the Hierarchical Timed Automata and the translated models. The proper control locations refer to basic states. The pseudo-states are entries and exits that translate into committed locations. The translation scales well and the introduced committed locations do not increase the state space since forks and joins trigger deterministic sequences of transitions. The extra auxiliary variables are tightly coupled to the current location and they do not have a major impact on the state space either.

6.3.2 Model-Checking the Translated UPPAAL Model

We used the translation as the input to the UPPAAL tool. We checked for deadlocks and the two desirable properties:

(i) `A[] ( heart_sub.FLATLINE => (wasSwitchedOff == 1) )`
(ii) `A[] ( heart_Sub.AfterAContraction => A<> heart_Sub.AfterVContraction )`

The system as described is not deadlock free: when the programmer terminates after switching off the pacemaker, and the heart stops beating, a configuration is reached where time can delay indefinitely. In one variation, the programmer was explicitly disallowed to exit. In a second variation, the pacemaker could not be switched off. In these two variations deadlock freedom was established[1]

Property *(i)* is a safety property and states that the heart never stops for too long unless the pacemaker is switched off by the programmer (in which case we cannot give any guarantees). Property *(ii)* is a response property and states, that after an atrial contraction, there will *inevitably*

[1] The verification time is in order of seconds. The exact value is not relevant here since there is nothing to compare with. It just shows the feasibility.

follow a ventricular contraction. In particular this guarantees, that no deadlocks are possible between these control situations. These properties hold though the validity of property *(i)* is strongly dependent on the parameter setting of the model. We use the following constants:

```
REFRACTORY_TIME          = 50
SENSE_TIMEOUT            = 15
DELAY_AFTER_V            = 50
DELAY_AFTER_A            =  5
HEART_ALLOWED_STOP_TIME = 135
MODE_SWITCH_DELAY        = 66
```

If the programmer is allowed to switch between modes very fast, it is possible that she prevents the pacemaker from doing its job. E.g., for `MODE_SWITCH_DELAY = 65` the property *(i)* does not hold any more. In practice it is often a problem to find parameter settings, that entail a safe or correct operation of the system. In related work, an extended version of UPPAAL is used to derive parameters yielding property satisfaction automatically, see [HRSV01].

Chapter 7

Conclusion

We defined a Hierarchical Timed Automata formalism targeted to UML and its formal syntax and semantics. We presented a translation to UPPAAL timed automata to show the feasibility of the analysis. The format of the model is in XML, making it suitable for UML and interchange with other tools.

The model has the advantages of UML modeling and the formalism of timed automata, which makes it appropriate for modeling large systems and analysing them. The model-checking technique hides the complexity of formal techniques and the only effort of the user is to draw the model and write the properties. The verification is automatic. The model is targeted to the UML standard, making it appealing for engineers. We are approaching our goal to spread more widely the use of formal techniques in a usable way.

Future work on Hierarchical Timed Automata is the analysis of the hierarchical model without translation. It is more difficult but the hierarchical model has more information that can be used to perform local search and reduce the state space. A hierarchical representation of DBM[1] to represent symbolic zones suits better the hierarchy. Local search in two unrelated parallel sub-states would reduce the state space. The hierarchy can guide a search path, making use of search cache more efficient. If a super-state can be searched independently, it can be flushed from memory with keeping only entry and exit pointers. These are only a few techniques that have to be explored.

[1] difference bound matrix

Part III

A Real-Time Animator for Hybrid Systems

Chapter 8

Hybrid Systems

8.1 Syntax

Let X be a set of real-valued variables X ranged over by x, y, z etc including a time variable t.

We use $\dot{x}$ to denote the derivative (rate) of x with respects to the time variable t. Note that in general $\dot{x}$ may be a function over X; but $\dot{t} = 1$. We use $\dot{X}$ to stand for the set of differential equations in the form $\dot{x} = f(X)$ where f is a function over X.

Assume a set of predicates over the values of X; for example, $2^x + 1 \leq 10$ is such a predicate. We use $\mathcal{G}$ ranged over by g, h etc to denote the set of boolean combinations of the predicates, called *guards*.

To manipulate variables, we use concurrent assignments in the form: $x_1 := f_1(X) \ldots x_n := f_n(X)$ that takes the current values of the variables X as parameters for f_i and updates all x_i's with $f_i(X)$'s simultaneously. We use Γ to stand for the set of concurrent assignments.

We shall study networks of hybrid automata in which component automata synchronize with each other via complementary actions. Let $\mathcal{A}$ be a set of action names. We use $\mathcal{A}ct = \{\ a? \mid \alpha \in \mathcal{A}\ \} \cup \{\ a! \mid \alpha \in \mathcal{A}\ \} \cup \{\ \tau\ \}$ to denote the set of actions that processes can perform to synchronize with each other, where τ is a distinct symbol representing internal actions.

A hybrid automaton over X, $\dot{X}$, $\mathcal{G}$, $\mathcal{A}ct$ and Γ is a tuple $\langle L, E, I, T, L_0, X_0 \rangle$ where

- L is a finite set of names standing for control nodes.
- E is the equation assignment function: $E : L \to 2^{\dot{X}}$.
- I is the invariant assignment function: $I : L \to \mathcal{G}$ which for each node l, assigns an invariant condition $I(l)$.
- T is the transition relation: $T \subseteq L \times (\mathcal{G} \times \mathcal{A}ct \times \Gamma) \times L$. We denote $(l, g, \alpha, \gamma, l')$ by $l \xrightarrow{g,\alpha,\gamma} l'$. For simplicity, we shall use $l \xrightarrow{g,\gamma} l'$ to stand for $l \xrightarrow{g,\tau,\gamma} l'$.
- $l_0 \in L$ is the initial node.
- X_0 is the initial variable assignment.

To study networks of automata, we introduce a CCS-like parallel composition operator. Assume that $A_1, ..., A_n$ are automata. We use $\overline{A}$ to denote their parallel composition. The intuitive meaning of $\overline{A}$ is similar to the CCS parallel composition of $A_1, ..., A_n$ with *all* actions being restricted, that is, $\overline{A} = (A_1 | ... | A_n) \backslash \mathcal{A}ct$. Thus only synchronization between the components A_i is possible. We call $\overline{A}$ a *network of automata*. We simply view $\overline{A}$ as a vector and use A_i to denote its ith component.

Example 1 In figure 8.1 we give a simple example hybrid automaton which describes a bouncing ball and a touch sensitive floor. The left automaton defines three variables, x, the horizontal distance from the starting point, the height y and the speed upwards u. Initially the x-speed is 1, the ball is at 20 m height and the gravitational constant is 9.8. The variables will change according to their equations until the transition becomes enabled when $y <= 0$. The middle automaton is a model of a sensor that will issue a signal when the ball hits the floor. The right automaton is on the UPPAAL side. It synchronizes with the sensor signal and resets a clock z. If the intervals between signals are longer than 5 time units it will return to the initial location, but the first interval that is shorter will lead to the location *low_bounces*.

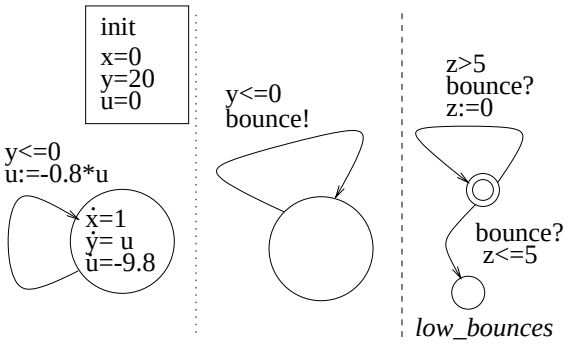


Figure 8.1: Bouncing ball with touch sensitive floor and control program.

Example 2 As a more complex example we show a model of an industrial robot. In figure 8.2 a schematic view of robot with a jointed arm is shown. The inner arm can be turned 360 degrees around the z-axis, it can also be raised and lowered between 40 and 60 degrees. The outer arm is positioned at the tip of the inner arm and can be raised and lowered.

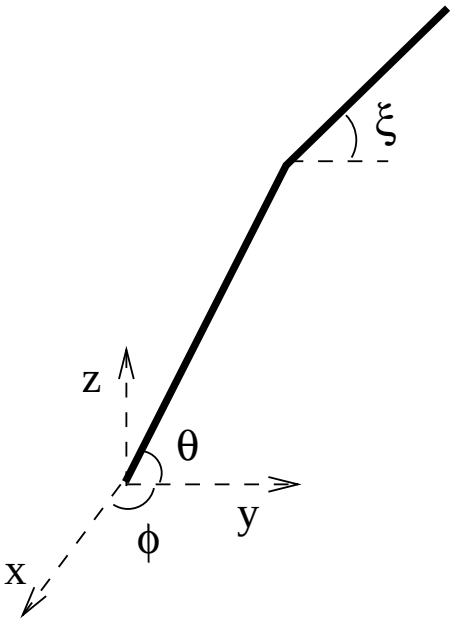


Figure 8.2: Industrial robot with three degrees of freedom.

In figures 8.3 and 8.4 the hybrid automaton controlling the motion of the robot is shown. We do not show the simple control automaton that starts and stops the execution.

When the execution starts the inner arm will stand still until it receives the `move2pickup?` signal from the control automaton. Then the arm will start turning and lowering the arm so that the gripping tool on the outer arm can reach a table where it will pick up an object. When th inner arm reaches a position in front of the table it will try to synchronize with the outer arms automaton on the signal `pickup!` . After the pickup the robot will raise and turn to another table where it will again try to synchronize with the outer arm on `release!` . After the release the robot will return to the original position and issue the `back!` signal to the controller.

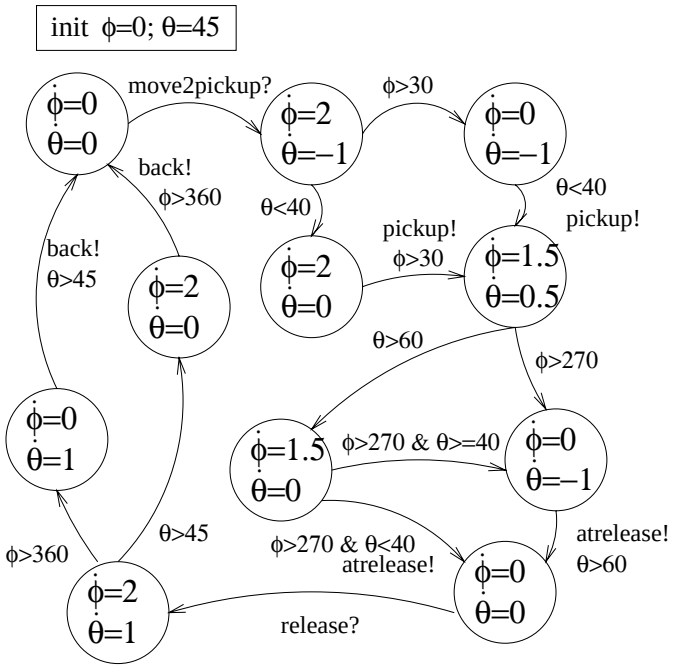


Figure 8.3: Robot inner arm automaton

The automaton, in figure 8.4, controlling the outer arm is simpler since the outer arm only has one degree of freedom. It starts with lowering the arm when the controller sends a `move2pickup2!`. When it comes to the right angle it will stop and wait for the inner arm to do the same, then they will synchronize on `pickup`. After the pickup the arm will raise until it reaches its max (30 degrees), then it will wait for the inner arm to reach the release position. When the robot is at the release table the outer arm will descend and then release the object in its grip. On the return to the original position the outer arm will rise again.

8.2 Semantics

To develop a formal semantics for hybrid automata we shall use variable assignments. A *variable assignment* is a mapping which maps variables X to the reals. For a variable assignment σ and a delay Δ (a positive real), $\sigma + \Delta$ denotes the variable assignment such that

$$(\sigma + \Delta)(x) = \sigma(x) + \int_{\Delta} \dot{x} dt$$

For a concurrent assignment γ, we use $\gamma[\sigma]$ to denote the variable assignment σ' with $\sigma'(x) = Val(e, \sigma)$ whenever $(x := e) \in \gamma$ and $\sigma'(x') = \sigma(x')$ otherwise, where $Val(e, \sigma)$ denotes the value

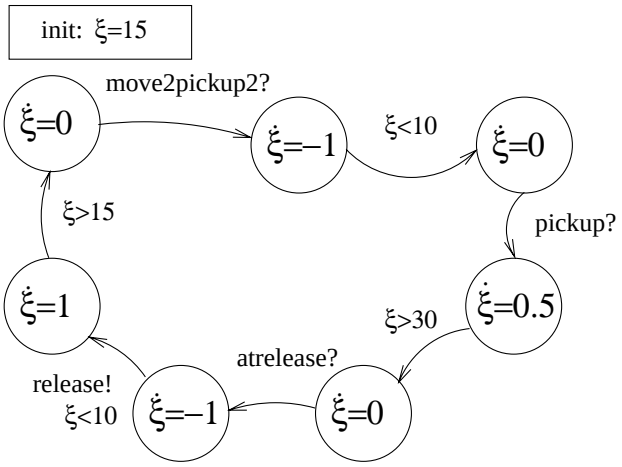


Figure 8.4: Robot outer arm automaton

of e in σ. Given a guard $g \in \mathcal{G}$ and a variable assignment σ, $g(\sigma)$ is a boolean value describing whether g is satisfied by σ or not.

A *node vector* $\overline{l}$ of a network $\overline{A}$ is a vector of nodes where l_i is a location of A_i. We write $\overline{l}[l_i'/l_i]$ to denote the vector where the ith element l_i of $\overline{l}$ is replaced by l_i'.

A *state* of a network $\overline{A}$ is a configuration $(\overline{l}, \sigma)$ where $\overline{l}$ is a node vector of $\overline{A}$ and σ is a variable assignment.

The *semantics of a network* of automata $\overline{A}$ is given in terms of a labelled transition system with the set of states being the configurations. The transition relation is defined by the following three rules:

- $(\overline{l}, \sigma) \overset{\alpha}{\rightsquigarrow} (\overline{l}[l_i'/l_i], \gamma_i[\sigma])$ if $l_i \overset{g_i\ \alpha,\gamma_i}{\longrightarrow} l_i'$ and $g_i(\sigma)$ for some l_i, g_i,α,γ_i.
- $(\overline{l}, \sigma) \overset{\tau}{\rightsquigarrow} (\overline{l}[l_i'/l_i, l_j'/l_j], (\gamma_j \cup \gamma_i)[\sigma])$ if $l_i \overset{g_i\ a!\ \gamma_i}{\longrightarrow} l_i'$, $l_j \overset{g_j\ a?\ \gamma_j}{\longrightarrow} l_j'$, $g_i(\sigma)$, $g_j(\sigma)$, and $i \neq j$, for some l_i, l_j, g_i, g_j, a, γ_i, γ_j.
- $(\overline{l}, \sigma) \overset{\Delta}{\rightsquigarrow} (\overline{l}, \sigma + \Delta)$ if $I(\overline{l})(\sigma)$ and $I(\overline{l})(\sigma + \Delta)$ for all positive real numbers Δ.

where $I(\overline{l}) = \bigwedge_i I(l_i)$.

The execution of a hybrid automata then becomes an alternating sequence of delay and action transitions in the form:

$s_0 \overset{\Delta_0}{\rightsquigarrow} (\overline{l}_0, \sigma_0 + \Delta_0) \overset{\alpha_0}{\rightsquigarrow} (\overline{l}_1, \sigma_1) \overset{\Delta_1}{\rightsquigarrow} (\overline{l}_1, \sigma_1 + \Delta_1) \overset{\alpha_1}{\rightsquigarrow} (\overline{l}_2, \sigma_2) \ldots (\overline{l}_i, \sigma_i) \overset{\Delta_i}{\rightsquigarrow} (\overline{l}_i, \sigma_i + \Delta_i) \overset{\alpha_i}{\rightsquigarrow} (\overline{l}_{i+1}, \sigma_{i+1})$

8.3 Tick semantics

The operational semantics above defines how an automaton will behave at every real-valued time point with arbitrarily fine precision. In fact, it describes all the possible runnings of a hybrid automata.

In practice, a "sampling" technique is often needed to analyse a system. Instead of examining the system at *every* time point, which often is impossible, only a finite number of time points are chosen to approximate the full system behavior. Based on this idea, we shall adopt a time–step semantics called δ–semantics characterized by the granularity δ, which describes how a hybrid system shall behave in every δ time units. In practical applications, the time granularity δ is chosen according to the nature of the differential equations involved. In a manner similar to sampling of measured signals the sampling interval should be short for rapidly changing functions. To achieve finer precision, we can choose a smaller granularity.

We use the distinct symbol χ to denote the sampled time steps. Now we have a discrete semantics for hybrid automata.

- $(\bar{l}, \sigma) \overset{\chi}{\mapsto} (\bar{l}, \sigma + \delta)$ if $(\bar{l}, \sigma) \overset{\delta}{\rightsquigarrow} (\bar{l}, \sigma + \delta)$ and
- $(\bar{l}, \sigma) \overset{\alpha}{\mapsto} (\bar{l}', \sigma')$ if $(\bar{l}, \sigma) \overset{\alpha}{\rightsquigarrow} (\bar{l}', \sigma')$

We use β_i to range over $\{\chi, \tau\}$ representing the discrete transitions. The "sampled" runnings of a hybrid automaton will be in the form:

$(\bar{l}_0, \sigma_0) \overset{\beta_1}{\mapsto} (\bar{l}_1, \sigma_1) \ldots (\bar{l}_i, \sigma_i) \overset{\beta_{i+1}}{\mapsto} (\bar{l}_{i+1}, \sigma_{i+1}) \ldots$

In the following section, we shall present a real time animator based on the δ-semantics. For a given hybrid automaton, the animator works as an interpretor computing the δ-transitions step by step using CVODE, a differential equations solver.

Chapter 9

Implementation

Our goal is to extend the UPPAAL tool to deal with hybrid systems. The UPPAAL GUI is written in Java and the differential equation solver that we have adopted, CVODE, is written in C. This gives the natural architecture of the animator: the animator itself with the objects is written in Java and the engine of the animator in C, connected through the Java native interface (JNI). The two main layers of the implementation are the animation system and the CVODE layers.

9.1 The Animation System Layer

The system to be modeled is defined as a collection of *objects*. Each object is described by a hybrid automaton with its corresponding variables. Every state of the hybrid automaton has a set of equations and transitions. The equations, conditions (guards) and assignments are given as logical/arithmetic expressions with ordinary mathematical functions such as sine, cosine ..., and also user defined functions.

The evaluation of the object equations, conditions and assignments is written in C. Each animator object, i.e. a hybrid process, is associated with one UPPAAL process that is an abstraction of the hybrid part and a bridge to UPPAAL. The abstraction is modeled as a stub process that performs the same synchronizations as the hybrid counterpart. This choice of implementation is motivated by the desire to model-check the rest of the UPPAAL processes as a closed system. Figure 9.1 shows the association of animator objects with UPPAAL automata.

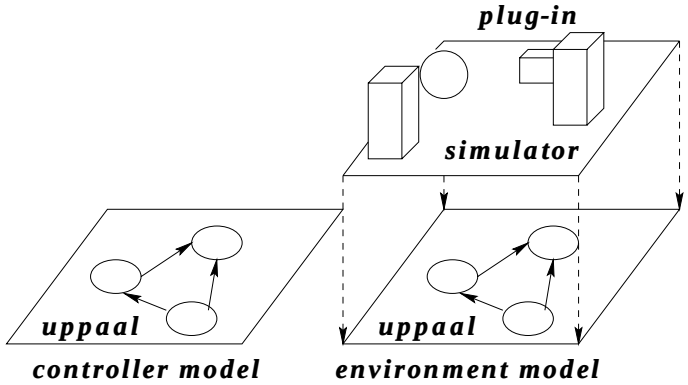


Figure 9.1: Association between animator objects and UPPAAL automata.

9.2 The CVODE Layer

At the heart of the animator we have used the CVODE [CH96] solver for ordinary differential equations (ODE's). This is a freely available ODE solver written in C, but based on two older solvers in Fortran.

The mathematical formulation of an initial value ODE problem is

$$\dot{x} = f(t, x),\ x(t_0) = x_0,\ x \in \boldsymbol{R}^N. \tag{9.1}$$

Note that the derivative is only first order. Problems containing higher order differential equations can be transformed to a system of first order. When using CVODE one gets a numerical solution to (9.1) as discrete values x_n at time points t_n.

CVODE provides several methods for solving ODE's, suitable for different types of problem. But since we aim at general usage of the animator engine we cannot assume any certain properties of the system to solve. Therefore we only use the full dense solver and assume that the system is well behaved (non-stiff in numerical analysis terminology). This will give neither the most memory efficient nor the best solution, but the most general.

We use one CVODE solver for the whole system. This is set up and started with new initial values at the beginning of each delay transition. The calculations are performed stepwise, one "tick" (δ–transition) at a time. After each tick all the conditions of the current state are checked, if any is evaluated to true one has to be taken. If an assignment on the transition changes a variable the solver must be reinitialized before the calculations can continue.

It is worth pointing out that the tick length δ is independent of the internal step size used by the ODE solver, the solver will automatically choose an appropriate step size according to the function calculated and acceptable local error of the computation. From the solvers point of view the tick intervals can be seen as observation or sampling points.

After each tick the system variables are returned to the Java side of the animator where they are used either to update a graph or as an input to move graphical objects.

Example 1, continued In figure 9.2 a plot of the system described in figure 8.1 is shown. The plot shows the height and the distance of the bouncing ball. Not shown in the figure is that the touch sensitive floor will create a signal every time the ball hits the floor, and that the system will continue running until the time between bounces is less than 1 second.

Example 2, continued For the robot example we only show, in figure 9.3, how the outer arm will raise and turn during the execution of the system. In order to get a better view, the plot only shows the movement from the initial position to the release. The robot starts at the right and turns counter clock-wise.

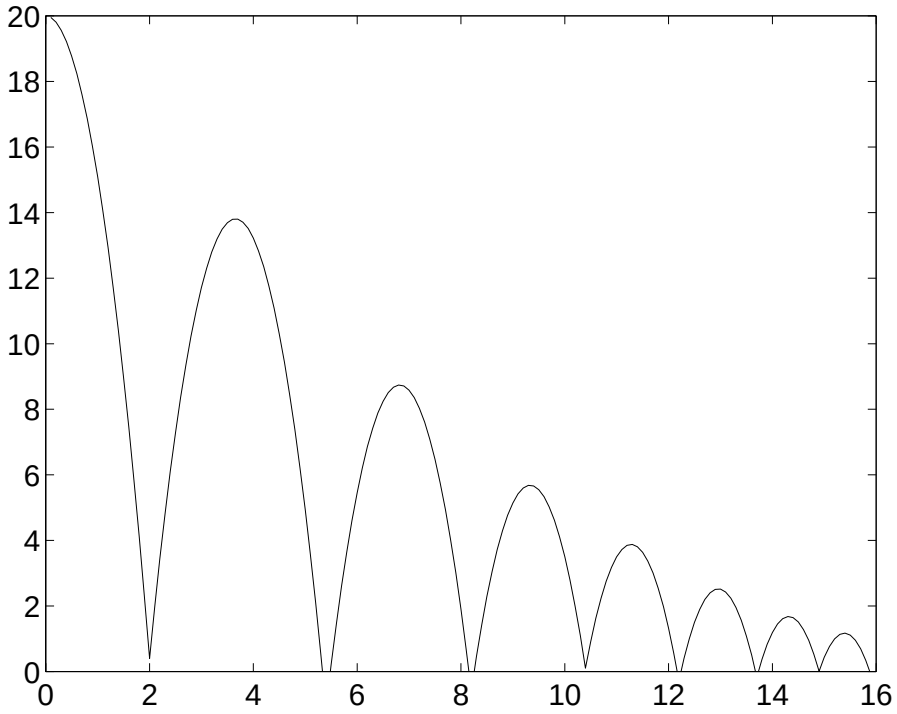


Figure 9.2: Bouncing ball on touch sensitive floor that continues until bounces are shorter than 1 second

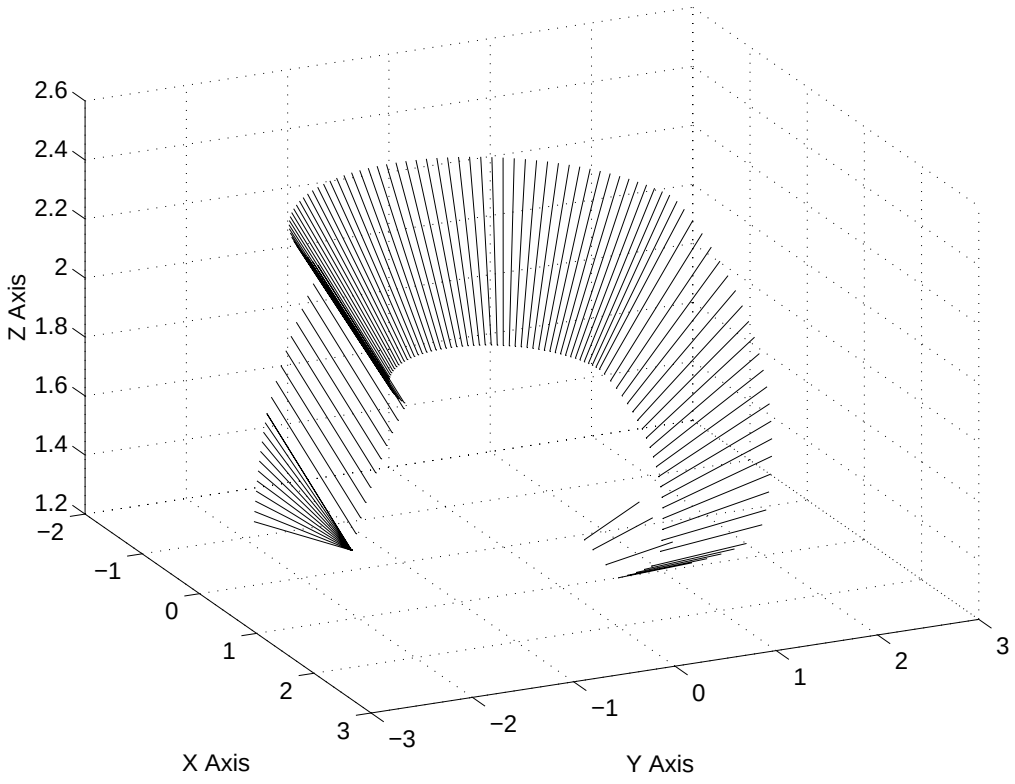


Figure 9.3: The movement of the robots outer arm.

Conclusion

We have presented a real time animator for hybrid automata. For a given hybrid automaton modeling a dynamical system and a given time granularity representing sampling frequency, the animator demonstrates a possible running of the system in real time, which is a sequence of sampled transitions. The animator has been implemented in Java and C using CVODE, a software package for solving differential equations. As future work, we aim at a graphical user interface for editing and showing moving graphical objects and plotting curves. The graphical objects act on the screen according to the differential equations and synchronize with controllers described as timed automata in UPPAAL.

Part IV

Appendix

Chapter 10

Figures

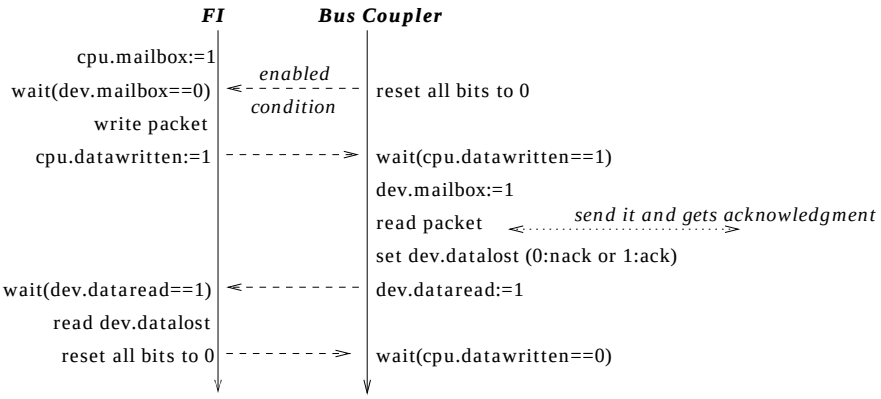


Figure 10.1: Communication protocol from the FI to the bus coupler.

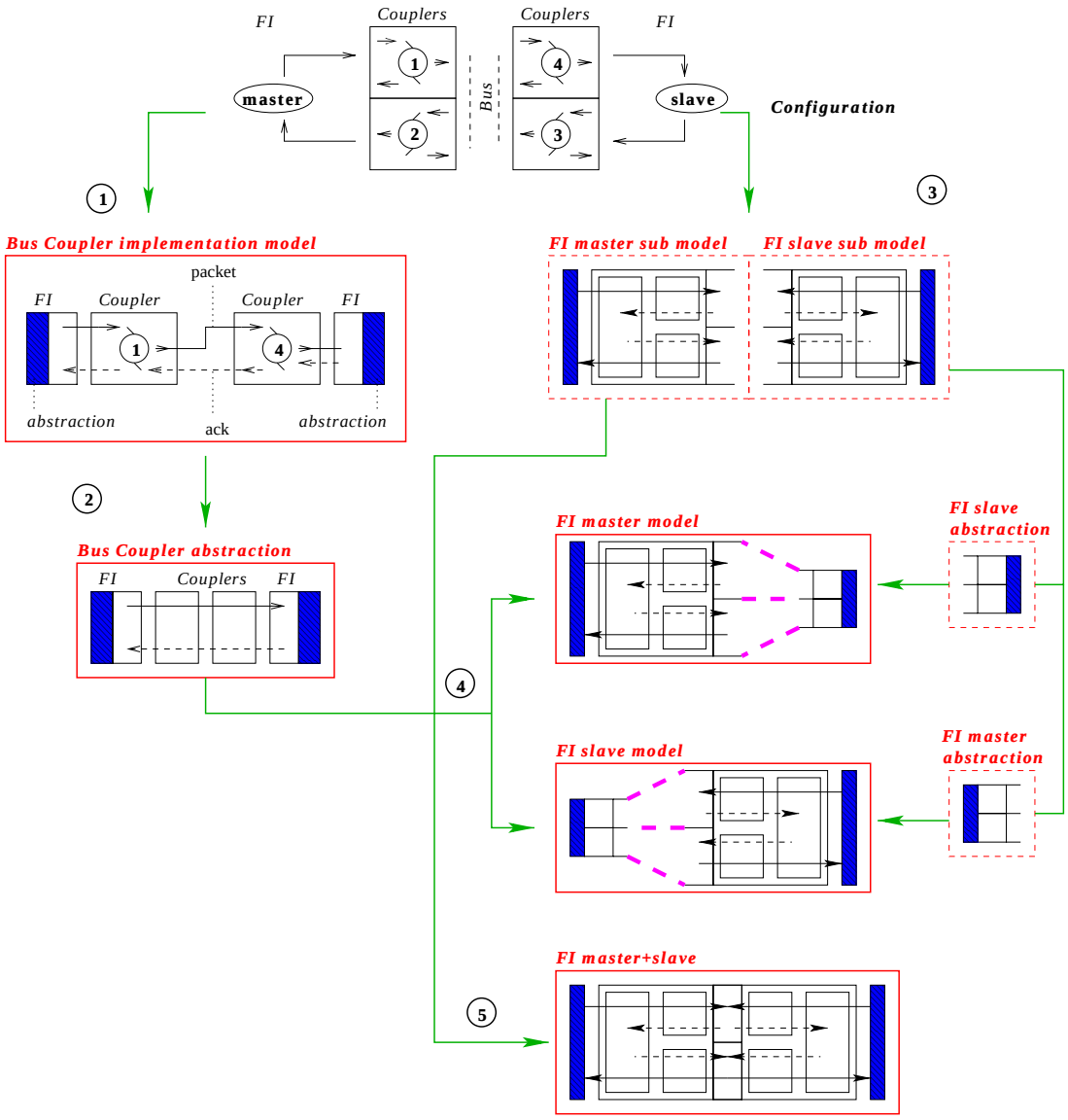


Figure 10.2: Modeling framework.

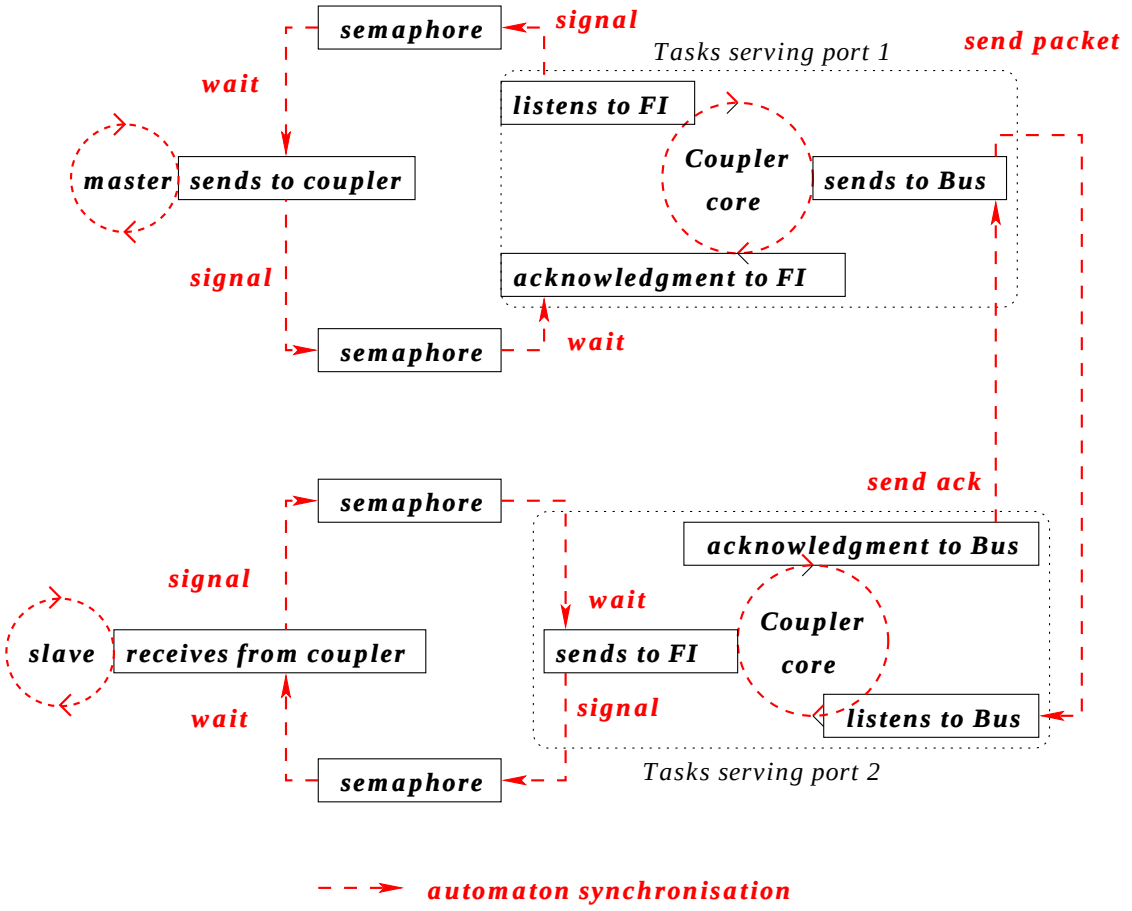


Figure 10.3: Static structures of the implementation model. Tasks are represented as circles and functions/semaphores as rectangles.

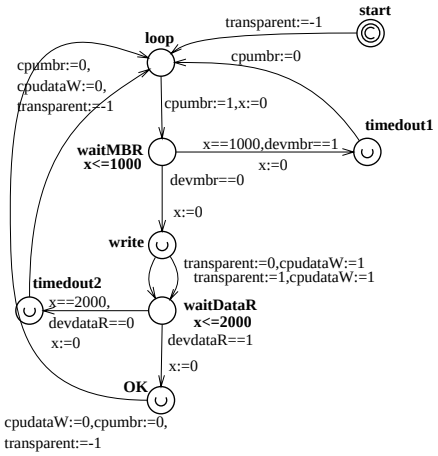


Figure 10.4: The template of the FI master.

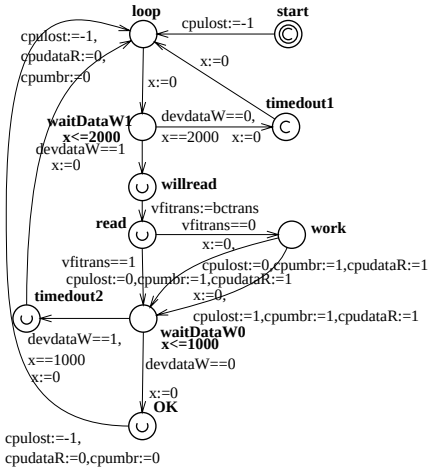


Figure 10.5: The template of the FI slave.

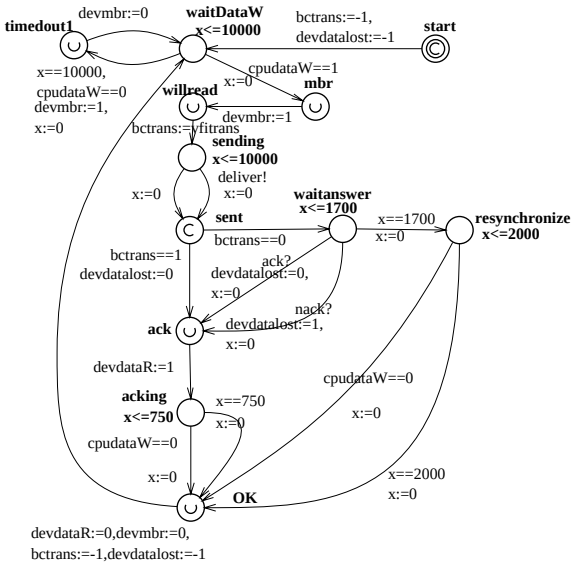


Figure 10.6: The template of the master coupler.

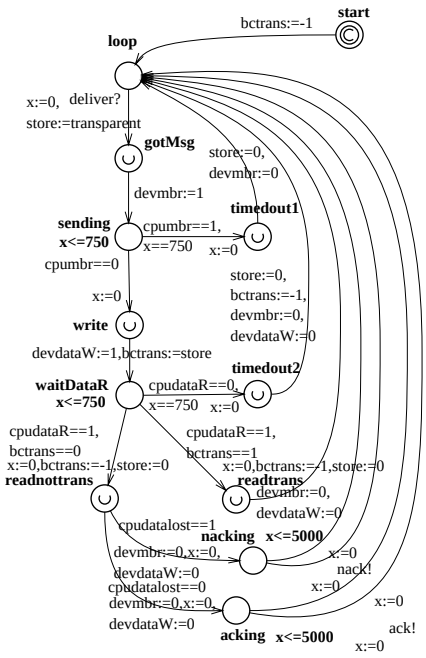


Figure 10.7: The template of the slave coupler.

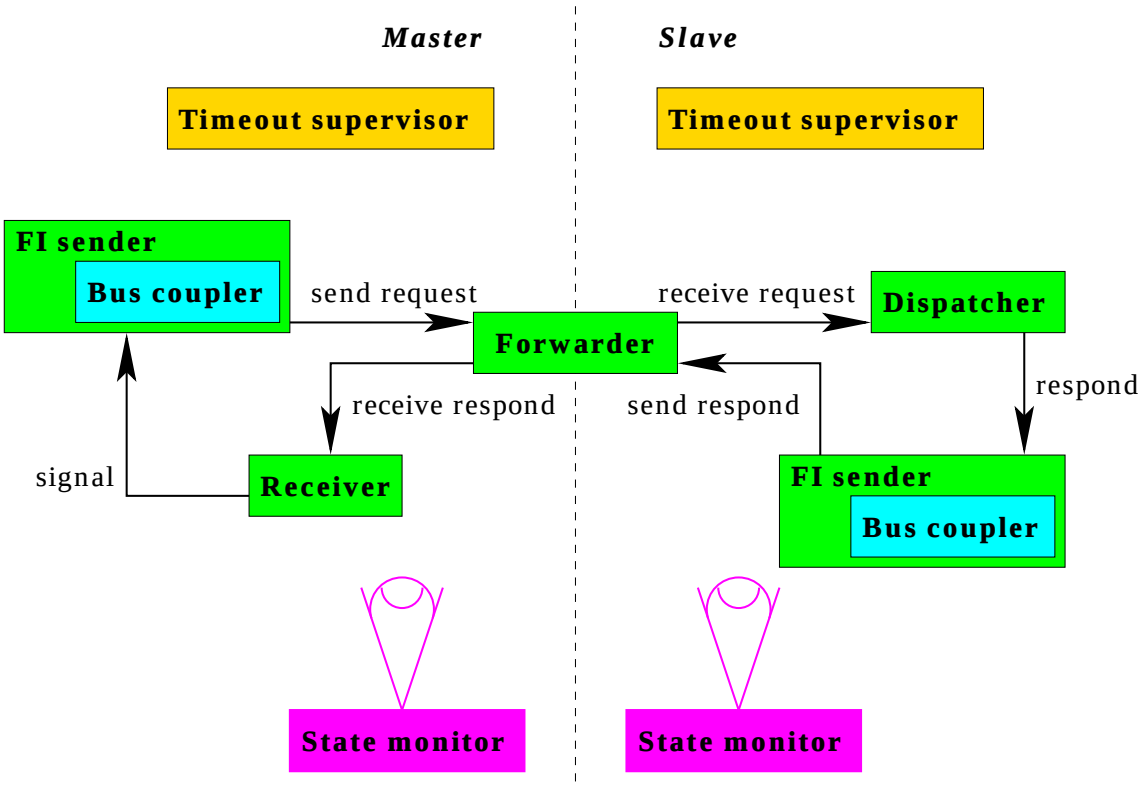


Figure 10.8: FI model overview.

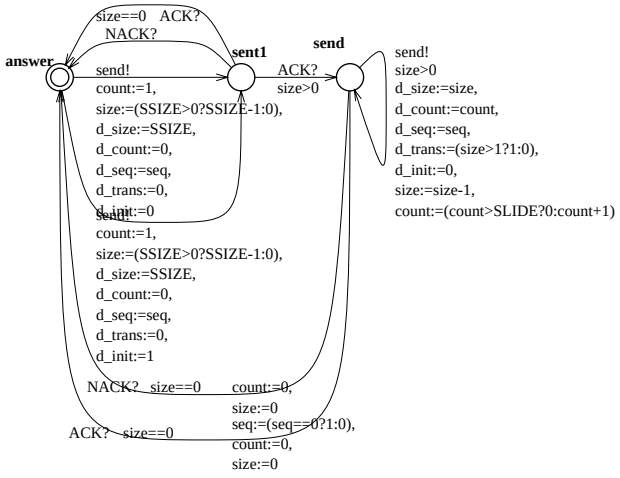


Figure 10.9: Slave test working with the master.

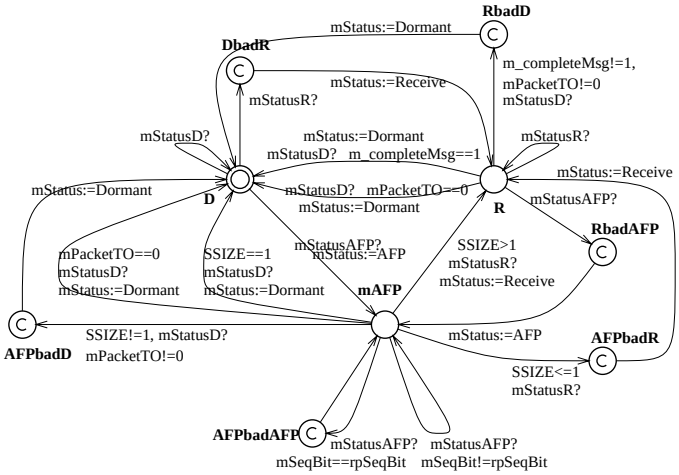


Figure 10.10: Master monitor automaton.

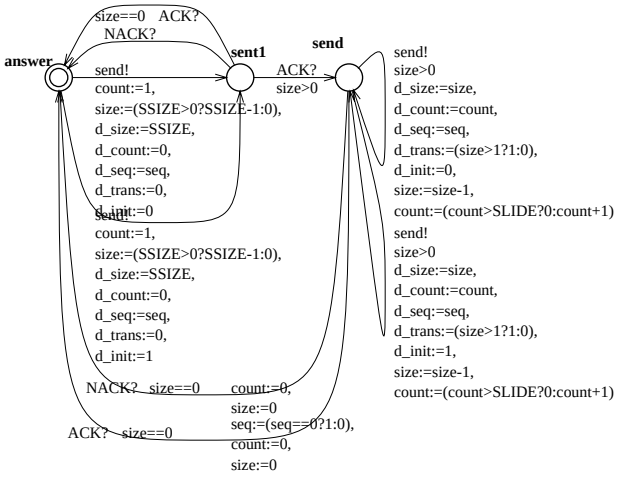


Figure 10.11: Master test working with the slave.

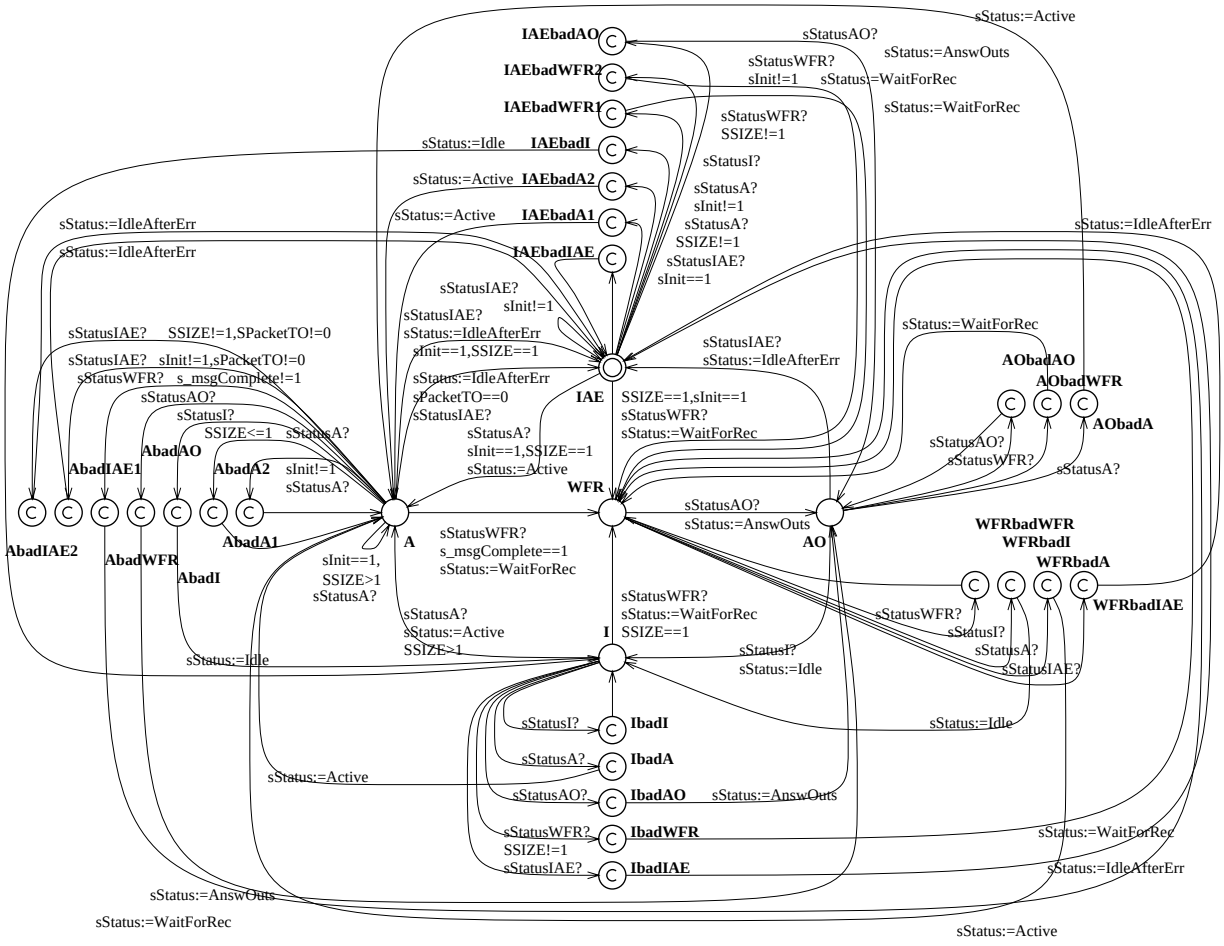


Figure 10.12: Slave monitor automaton.

Chapter 11

Translation Algorithms

Algorithm: *PHASE I: instantiateTemplates*

input: Stack S of superstates to translate
output: Set T of (flat) timed automata
Set GJ of global join starting points

$T := \{$*Global_Kickoff* automaton for $s \in S\}$
$GJ := \varnothing$
WHILE *notempty*(S)
  $I :=$ *pop*(S)
  $\mathcal{C} :=$ {non-basic locations in I}
  FORALL $c \in \mathcal{C}$
    push$(S, [c$ in $I])$
    /$\star$ $[c$ in $I]$ inherits all invariants attached to I $\star$/
    create a location $\hat{c}$ in $\hat{I}$
    $E_c :=$ {set of entries of c in I}
    FORALL $e \in E_c$
      create a committed location $\hat{c_e}$ in $\hat{I}$
      create a transition from $\hat{c_e}$ to $\hat{c}$ in $\hat{I}$
      /$\star$ this transition carries a synchronization "`enter_`$\hat{c}$`_via_`e`!`" $\star$/
    IF *type*$(I) =$ XOR THEN
      $GJ := GJ \cup \{c$ in $I\}$

  $T := T \cup \{$translation $\hat{I}$ of superstate I, depending on *type*$(I)\}$

Algorithm: *PHASE II: expandGlobalJoins*
input: Set GJ of global join starting points
output: Auxiliary constructions: counters and guarded transitions

$JoinTrees := \varnothing$
FORALL $gj \in GJ$
collect all trees t of control locations, that can synchronize to gj;
the leaves of t are sets of basic locations, that share transitions to exits x.
/* These sets are singletons, if x is an ordinary exit
and span over all basic locations in the superstate otherwise */
$JoinTrees := JoinTrees \cup \{t\}$
FORALL $tree \in JoinTrees$
let $\hat{L} := \{\hat{l} \mid l \text{ is element in a basic location set of } tree\}$
declare the counter $trigger_{tree}$
FORALL $\hat{l} \in \hat{L}$
FORALL transitions $\hat{k} \to \hat{l}$
add the assignment $trigger_{tree} := trigger_{tree} + 1$ to $\hat{k} \to \hat{l}$
FORALL transitions $\hat{l} \to \hat{m}$
add the assignment $trigger_{tree} := trigger_{tree} - 1$ to $\hat{l} \to \hat{m}$
let $N :=$ number of leaf sets in $tree$
let $\mathcal{S}_{tree} :=$ substates occurring in $tree$
FORALL transition t starting at $root(tree)$
create a chain of transitions, starting with $\hat{t}$,
corresponding to exiting every $s \in \mathcal{S}_{tree}$
/* see Figure 6.2 (b); note the additional guard $trigger_{tree}$==N */

Algorithm: *PHASE III: postprocessChannels*
input: *priorityQueue* Q over (*syncSignal*, *transition*, *instantiation*)

WHILE notempty(Q)
(*syncSignal*, *transition*, *instantiation*) $:= pop(Q)$
IF $\exists$ transition t with $match(syncSignal)$ in *instantiation*:
create a new channel c
replace $channel(syncSignal)$ on *transition* by c
FORALL transitions t' with $match(syncSignal)$ outside *instantiation*
create a copy of t', where $channel(syncSignal)$ is replaced by c
if there is an entry of t' in Q, add an entry for the created copy to Q

Chapter 12

Glossary

configuration A configuration is a snapshot of the system, where every location is either active or inactive and every variable and clock is set to *one specific* value. A configuration is *proper*, if all active basic locations are proper.

entry A pseudo-location, that is passed to activate the corresponding superstate.

entry point Copy of an entry, displayed as bullet (•), annotated with the name of the entry.
This is a notational/graphical convenience with the semantics of an alias.

exit A pseudo-location, that is passed to inactivate the corresponding superstate.

exit point Copy of an exit, displayed as bullseye (⊙).
This is a notational/graphical convenience with the semantics of an alias.

fork Auxiliary structure used in AND superstates.
A fork connects an entry of the superstate with the entries of the parallel substates, thus activating them.

global join Synchronous exit of various parallel superstates.
A global join gives rise to a *tree* of joins (connected via exits), that is specific to a *root transition* (the one that is executed immediately after the join).
A global join can only be started, if all participants can synchronize on their exit. It is executed without interruption, including the execution of the root transition. (In the UPPAAL translation, this requires special constructions, see Section 6.2.2.)

join Auxiliary structure used in AND superstates.
A join connects exits from each of the enclosed superstates with an exit of the superstate itself.

location The basic unit of control.
A location can be *basic* or a *superstate*, i.e., itself a hierarchical timed automaton. Basic locations are either *proper* or *pseudo-locations*. At any time, a location is either *active* or *inactive*.

pseudo-location Auxiliary location to encode complex transitions.
Though physically a (committed) location, this does usually not correspond to a state the modeled system can be in and exists solely for modeling purposes, typically to encode forks, joins, or multi-synchronization.

pseudo-transitions Auxiliary transition to encode a part of a run-to-completion step, e.g., to encode entry, exit, or multi-synchronization.
Pseudo-transitions are connected to at least one committed location. Restrictions on allowed guards, assignments, and synchronizations apply.

pre-exit A location that has a transition to an exit.

run-to-completion step A sequence of transitions, containing one proper transition and arbitrary many pseudo-transitions.
This amounts to a macro-transition leading from one proper configuration to to a subsequent proper configuration.

superstate A non-basic location.

We distinguish XOR superstates (exactly one of the substates is active, if the superstate is active) and AND superstates (parallel composition: all substates are active, if the superstate is active).

transition A transition connects two locations, carrying guards, assignments, and synchronization. If these are non-basic, the transition connects to specific entries or exits. A transition is either *proper* or *pseudo*.

Bibliography

[ABB+] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, , and Wang Yi. UPPAAL - Now, Next, and Future. To appear in Proceedings of the Summer School on Modelling and Verification of Parallel Processes (MOVEP'2k), Nantes, France, June 19 to 23, 2001. Available at `http://www.docs.uu.se/~paupet/`.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. In *Theoretical Computer Science*, number 125, pages 183–235, 1994.

[ADY00] Tobias Amnell, Alexandre David, and Wang Yi. A real time animator for hybrid systems. In J. Davidson and S.L. Min, editors, *Languages, Compilers, and Tools for Embedded Systems ACM SIGPLAN Workshop LCTES 2000, Vancouver, Canada, June 18, 2000, Proceedings*, volume 1985 of *Lecture Notes in Computer Science*. Springer–Verlag, June 2000.

[AKY99] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating Hierarchical State Machines. In *Proc. of the 26th International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 169–178. Springer–Verlag, 1999.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[CH96] S. Cohen and A. Hindmarsh. Cvode, a stiff/nonstiff ode solver in c. 1996.

[DM01] Alexandre David and M. Oliver Möller. From Hierarichcal Timed Automata to UPPAAL. Research Series RS-01-11, BRICS, Department of Computer Science, University of Aarhus, March 2001.

[Dou99] Bruce Powel Douglass. *Real-Time UML, Second Edition - Developing Efficitnt Objects for Embedded Systems*. Addison-Wesley, 1999.

[Dou00] Bruce Powel Douglass. *Doing Hard Time*. Adison Wesley, 2000.

[DY00] Alexandre David and Wany Yi. Hierarchical Timed Automata. unpublished draft, dated: April 23. Contact the authors `adavid@DoCS.uu.se`, `yi@DoCS.uu.se`, 2000.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 1987.

[Hen] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic on Computer Science*, LICS 96, pages 278–292.

[HG97] David Harel and Eran Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 7(30):31–42, July 1997.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Soft. Eng. Method 5:4*, oct 1996.

[HNSY94] Thomas. A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994.

[HRSV01] Thomas S. Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear parametric model checking of timed automata. Research Series RS-01-5, BRICS, Department of Computer Science, University of Aarhus, January 2001. 44 pp.

[HSLL97] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal Modelling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, pages "2–13". IEEE Computer Society Press, December 1997.

[KGLY95] Paul Pettersson Kim G. Larsen and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of the 10th International Conference on Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88. Springer–Verlag, 1995.

[KrJKW] Paul Pettersson Kå re J. Kristoffersen, Kim G. Larsen and Carsten Weise. Experimental batch plant - vhs case study 1 using timed automata and uppaal.

[LP97] Henrik Lönn and Paul Pettersson. Formal verification of a tdma protocol start-up mechanism. In *Proc. of IEEE Pacific Rim International Symposium on Fault-Tolerant Systems*, pages "235–242", 1997.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LPY98] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear Controller. In *Proc. of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems.*, volume 1384 of *Lecture Notes in Computer Science*, pages "281–297". Springer–Verlag, 1998.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[pro] Prover homepage: `http://www.prover.com`.

[Rha] Rhapsody is a commercial product of I-Logix. Documentation and whitepapers are available from `http://www.ilogix.com/quick_links/white_papers/index.cfm`.

[RT] P.R. D'Argenio J.-P. Katoen T.C. Ruys and J. Tretmans. The bounded retransmission protocol must be on time! In *Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1217, pages 416–431.

[Tan81] A.S. Tanenbaum. *Computer networks.* Prentice–Hall, 1981.

[Vis] VisualState™ is a commercial product of IAR Systems. Detailled information is available from `http://www.iar.com`.

Licentiate theses from the Department of Information Technology

2001-001 Erik Borälv: *Design and Usability in Telemedicine*

2001-002 Johan Steensland: *Domain-based partitioning for parallel SAMR applications*

2001-003 Erik K. Larsson: *On Identification of Continuous-Time Systems and Irregular Sampling*

2001-004 Bengt Eliasson: *Numerical Simulation of Kinetic Effects in Ionospheric Plasma*

2001-005 Per Carlsson: *Market and Resource Allocation Algorithms with Application to Energy Control*

2001-006 Bengt Göransson: *Usability Design: A Framework for Designing Usable Interactive Systems in Practice*

2001-007 Hans Norlander: *Parameterization of State Feedback Gains for Pole Assignment*

2001-008 Markus Bylund: *Personal Service Environments — Openness and User Control in User-Service Interaction*

2001-009 Johan Bengtsson: *Efficient Symbolic State Exploration of Timed Systems: Theory and Implementation*

2001-010 Johan Edlund: *A Parallel, Iterative Method of Moments and Physical Optics Hybrid Solver for Arbitrary Surfaces*

2001-011 Pär Samuelsson: *Modelling and control of activated sludge processes with nitrogen removal*

2001-012 Per Åhgren: *Teleconferencing, System Identification and Array Processing*

2001-013 Alexandre David: *Practical Verification of Real-time Systems*

UPPSALA
UNIVERSITY

Department of Information Technology, Uppsala University, Sweden