Code Synthesis for Timed Automata

Tobias Amnell

3rd October 2003

Abstract

In this thesis, we study executable behaviours of timed models. The focus is on synthesis of executable code with predictable behaviours from high level abstract models. We assume that a timed system consists of two parts: the control software and the plant (i.e. the environment to be controlled). Both are modelled as timed automata extended with real time tasks. We consider the extended timed automata as design models.

We present a compilation procedure to transform design models to executable code including a run-time scheduler (run time system) preserving the correctness and schedulability of the models. The compilation procedure has been implemented in a prototype C-code generator for the brickOS operating system included in the Times tool. We also present an animator, based on hybrid automata, to be used to describe a simulated environment (i.e. the plant) for timed systems. The tasks of the hybrid automata define differential equations and the animator uses a differential equations solver to calculate step-wise approximations of real valued variables. The animated objects, described as hybrid automata, are compiled by the Times tool into executable code using a similar procedure as for controller software.

To demonstrate the applicability of timed automata with tasks as a design language we have developed the control software for a production cell. The production cell is built in LEGO® and is controlled by a Hitachi H8 based LEGO® -Mindstorms control brick. The control software has been analysed (using the Times tool) for schedulability and other safety properties. Using results from the analysis we were able to avoid generating code for parts of the design that could never be reached, and could also limit the amount of memory allocated for the task queue.

Acknowledgements

First of all I would like to thank my supervisors: Wang Yi and Paul Pettersson for their advises and valuable comments on this thesis. I also would like to thank the other members, and former members, of the Design and Analysis of Real-Time Systems (DARTS) group, especially Alexandre David, Johan Bengtsson, Elena Fersman, Leonid Mokrushin and Fredrik Larsson. They have all contributed to a very inspiring research group. Our collaborative work and interesting discussions have been much fun and very rewarding.

I also would like to thank my colleagues at the DoCS subdepartment of the department of information technology. The informal coffee-room and lunch discussions have really broadened my insights in computer systems and many other subjects.

This work has been supported by the ASTEC competence centre in Uppsala. Without this support this work would not have been possible.

A special thanks goes to professor Björn Karlsson who, even though he may not be aware of it, inspired me to start as a PhD-student.

Finally I would like to thank my family: my mother Anna, my father Gösta, my twin sister Lovisa and my brother Mattias. Their love and support have been of great help in my life.

Publications

This thesis is based on work, parts of which have previously been published in these papers:

[A] Tobias Amnell, Elena Fersman, Paul Pettersson, Wang Yi, and Hongyan Sun. Code synthesis for timed automata. In *Nordic Journal of Computing*, volume 9 number 4, pages 269-300, 2002.

[B] Tobias Amnell, Alexandre David, and Wang Yi. A Real-Time Animator for Hybrid Systems. In proceedings of the *6th ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'2000)*, volume 1985 of Lecture Notes in Computer Science, pages 134-145, 2000.

[C] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - A Tool for Modelling and Implementation of Embedded Systems. In Proceedings of the *7th international Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 460ff, 2002.

Comments on my participation

In paper [A] the code generation for timed automata with tasks was presented. My contribution was the deterministic semantics for timed automata with tasks, implementation of a code generator for legOS and participation in the design of the controller of the production cell.

Paper [B] presented an earlier implementation of the animator for hybrid automata. The implementation was a collaboration between myself and Alexandre David. The animator presented in this thesis is a complete re-implementation by myself for the Times tool.

Paper [C] presents the Times tool and my contribution was the sections on code generation.

Other publications

I have also contributed to the following papers:

- Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems . To appear in Proceedings of the *1st International Workshop on Formal Modelling and Analysis of Timed Systems (FORMATS)*, 2003.

- Tobias Amnell, Alexandre David, Elena Fersman, Oliver Möller, Paul Pettersson, and Wang Yi. Tools for Real-Time UML: Formal Verification and Code Synthesis. In Proceedings of the *Workshop on Specification, Implementation and Validation of Object-oriented Embedded Systems (SIVOES 2001)*, 18-22 June 2001.

- 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. In Proceedings of *Modelling and Verification of Parallel Processes (MOVEP'2k)*, Nantes, France, June 19 to 23, 2000. *LNCS Tutorial 2067*, pages 100-125, 2001.

- Tobias Amnell and Pontus Jansson. Formal Modelling of a Central Lock System. In Proceedings of *Workshop on Real-Time Tools*, Aalborg University, Denmark, August 20, 2001.

Contents

Chapter 1

Introduction

This thesis presents an approach to the automation of software development for *real-time embedded systems*. In particular, it defines a deterministic semantics and presents a code generator for the design language *timed automata with tasks*, henceforth abbreviated TAT. The code generation uses previous results in schedulability analysis for TAT designs to generate executable software for real-time embedded systems with verified behaviour. Based on the code generator we have also developed an animator for hybrid automata. The animator is intended to be a simulation environment (or plant to be controlled) for controller designs where objects in the environment of the controller is modelled as hybrid automata. We also present a case study where the control software for a model of an industrial production cell has been designed, analysed and synthesised using the presented techniques.

The presented work is part of a larger effort in the group "Design and Analysis of Real-Time Systems" at the Department of Information Technology at Uppsala University. Our overall goal is to build a tool called Times for the development of verified and validated embedded real-time software. We build on the knowledge of verifying timed systems gained by developing the timed-automata model checker Uppaal [BLL$^+$96]. In Times we extend the work on verification of real time systems with code generation and analysis of schedulability, timing, resource utilisation etc.

1.1 Background

The rapid development of computers in the last decades have created computerised tools to be used in more and more areas of society. The earliest uses of computers, for information processing in for example banking and scientific computing, used huge machines and required skilled operators to run. Today computers exist in every day products such as cars, mobile phones and medical equipment, and no special computer skills are needed.

Many of these applications are not even seen as computers. They are instead examples of *embedded systems*. That is, systems where the computer is an integrated part of another device. In fact, more than 95% of the computers built today are embedded in such systems.

Many embedded systems are also examples of *real-time systems*. Real-time systems are computerised systems characterised by that the timing of the computed results are as important as the values themselves. The timing requirements are often imposed by the environment that the control software of the real-time system must interact with. The software must give timely responses to stimuli such as key-presses or arrival of network packages, otherwise undesired situations may occur.

A multimedia system decoding a video stream arriving on a network is an example of a system with real-time properties. If the system cannot perform decoding of the stream in a timely manner the perceived quality of the output will be decreased. So in this case quality requirements can be transformed into timing requirements and thereby qualifying the multimedia system as a real-time system. Digital control of an airbag in a car is another example of a real-time system where the late arrival of a value could have catastrophic results.

These two examples also serve as examples of *soft* and *hard* real-time systems respectively. The distinction between the two classes being the severity of the consequences of an unsatisfied timing requirement. In the multimedia case nothing worse than a low quality output will happen, but in the airbag example the missed deadline could result in severe injury or death. In this thesis we will discuss methods that provide guarantees needed in hard real-time systems.

As processors have become cheaper and more powerful new functions, previously unimagined, have been included in embedded systems. Much of the new functionality is realised through increasingly more complex software. At the same time the rapid technological development and the competitive market demands shorter and shorter time to develop the software. This means that new more efficient software development methods are needed.

Software for embedded real-time systems is usually developed using ordinary programming languages. For low end micro controllers assembly programming is still used, but for larger systems high level languages are more common. The most popular is probably C. But, due to the special nature of embedded systems many traditional development methods are not suitable or needs modifications to be applicable to the development embedded systems. For example, the hardware is often limited in speed and main memory and a secondary storage is seldom included. This means that the size of the software must be small to fit in the memory and the code must be fast so that the system performance is still sufficient.

Real-time requirements put further restrictions on the software. A necessary prerequisite for all methods that provide real-time guarantees is that the execution time of the programs is known. The problem of finding the worst case execution time of a program is complicated by many features of modern processors such as caches, pipelines and interrupts. Therefore the processor in an embedded real-time system is often of simpler types lacking complex features but with more predictable executions times. It may also be necessary to restrict the programming constructs to those that can be easily analysed. For example using constant bounds on loops.

To meet the challenges of embedded systems new tools are needed that can make the software development easier and help the software live through several generations of hardware platforms. Many CASE tools are emerging that can help with design, analysis and even implementation of embedded systems software. In this thesis we will present one such tool, Times.

1.2 Model checking

Embedded real-time systems are often employed in safety-critical applications, such as medical equipment and automotive systems. In such systems the correctness of the software, and the overall system, is of vital importance. But also less safety-critical devices may be critically dependent on the correctness of the software.

Traditional methods to ensure correctness of software systems include code reviews and testing. For embedded systems, which by definition are embedded into some other device, debugging and testing can be complicated and incomplete. Compared to a desktop system the interface to the system is limited and the program state can be hard to access. But the correctness of embedded systems are possibly even more important than for desktop systems. Correctiong an error in a running embedded system can be very costly or even impossible. Imagine for example the costs involved in repairing a software bug in thousands of cars already delivered to customers.

Traditional testing methods may be applied rigorously but it lies in their nature that they can only provide statistical measures of the correctness. To achieve firmer guarantees we need to apply methods based on mathematical techniques, so called *formal methods*. The term formal methods is used to collectively name mathematically-based languages, techniques and tools for specifying and verifying both software and hardware systems, see for example [CWA+96] for a survey.

Specification using a formal language is a formal method in its own right, but can also be seen as a prerequisite for formal verification techniques. *Formal verification* methods are used to analyse a system for desired properties, such as absence of deadlocks. To apply a formal verification method the system and the properties are expressed using formal languages allowing application of rigorous mathematical methods. The application of the method can then provide a proof that the system satisfies (or dose not satisfy) the properties.

One of the most successful verification techniques is *model checking* where the proof that the model satisfies a requirement is constructed automatically by a tool. In model-checking a finite model of the system is constructed and the tool performs a search of the state space of the model to determine if the properties are satisfied. The major challenge in model checking is to handle the large state spaces generated. There are several approaches to attack the state explosion problem e.g., by symbolic techniques [JEK+90], efficient data structures that reduce the memory consumption [Ben02], abstraction techniques [CGL94] and compositional methods [ELK89].

During the last decade many model checkers have been developed for different purposes, see Yahooda[1] for a collection of such tools. The model-checking technique has been very successful in verification of hardware circuit designs. Model checkers such as SMV [McM93] that uses Binary Decision Diagram [Bry86] for symbolic representation of the state space has been applied to verify large hardware systems [JEK+90]. On the other hand model checking of soft-

[1]Yahoda `http://anna.fi.muni.cz/yahoda/`

ware has been less successful due to the more complex structures and dynamic behaviour of software. Still there are some notable applications, we mention SPIN [Hol91, Hol97] as an example of a model-checker for asynchronous software systems, especially communication protocols. In the area of real-time systems we mention two model checkers for the timed automata, Uppaal [LPY97, ABB+01] and KRONOS [DOTY95, Yov97, BDM+98], and for hybrid automata the model checker HYTECH [HHWT97].

1.3 Software Synthesis

A problem with formal specification and verification techniques is that the verified system is usually not the actual implementation but an abstract model of it. To make use of the formal proof the relation between the implementation and the model must be defined. We may think of two possible approaches to overcome this problem; one starting in the implementation and one in the model.

In the first approach the actual implementation of an existing system is studied and abstraction techniques can be applied to get an abstract model that can be handled by the verification methods.

The second approach is to start from a model considered to be the design of a system to be constructed. The design model is verified first and then the verified model is transformed automatically to executable code. If the transformation steps are well defined the generated code will be correct-by-construction. For example, for telecommunication systems there exits well developed tools for code generation [Tel03] from specifications. For timed systems on the other hand, it has been a challenge to synthesise executable code from a timed model that guarantees the timing constraints imposed on the model.

In this thesis we adopt timed automata as design models for timed systems, and study how to transform such models to code. We will develop a code generator for an extended version of timed automata with data variables and executable tasks triggered by discrete transitions in the automata. We consider systems consisting of two parts:

- a control software, and
- a plant (the environment to be controlled).

We will use the TAT language to describe the control software as extended timed automata. Design models in TAT can be analysed using model-checking techniques to find safety properties, including schedulability of the released tasks and boundedness of the task queue.

The main problem we want to solve is how to generate executable code that preserves the safety properties of a design model. To achieve this we need to resolve the non-determinism in a TAT design model. The generated code also needs to interact with the underlying operating system to schedule the tasks for execution and to access sensors and actuators in the hardware.

The behaviour of the system is determined by the interaction between the control software and the plant it controls. When analysing the system using model-checking we will model the plant as timed automata with tasks. A real environment will have a richer set of behaviours than can be described using timed automata. Therefore we will extend the code generator so that the plant can be modelled using hybrid automata. Hybrid automata enables plant models with continuous behaviour, described as differential equations, combined with discrete actions. Generated code from both parts of the system can then be combined into an executable animator of the system.

The animator can be used for simulations and test executions of the system so that the designer may validate that the design does the right thing when executed in a more realistic environment. Several other modelling tools, such as Rhapsody[IL03] for UML and Telelogic TAU[Tel03] for SDL, provide similar automatic generation of executable prototypes that can be used to demonstrate the system behaviour before it is actually built. Compared to our animator the mentioned systems do not provide integrated code generation for both the control software and the environment.

Related Work

For digital signal processors (DSP:s) there has been a significant development of efficient software synthesis tools, see [BML99] for a survey of techniques. This specific application area lends itself to be described using data flow languages, such as synchronous data flow (SDF) [LM87]. In SDF a program is a directed graph where the vertexes are actors (computations) and the edges represent buffers. The software synthesis problem for such programs is to find a schedule for the actors that fulfils the partial order defined by the graph and that minimises some measure, such as memory usage, needed buffer length etc. The actors are described using assembly code or a programming language such as C.

Another well understood type of language that is used for software synthesis is based on FSM:s. Finite state machines are well suited to describe "control-dominated" embedded systems where input events result in reactions that produce output events. We also call such systems reactive systems.

In [BCG+99] Balarin et al. presented an approach for software synthesis based on *co-design FSM:s,* an extension to ordinary FSM with events and limited data variables. The authors describe how to transform a design to C-code while performing optimisations on code size using boolean function optimisations.

Another area where finite state machines are a natural description formalism is in specification of communication protocols. We mention the work presented in [CDO97] where an Esterel specification of a communication protocol is transformed into automata which are compiled into C using several optimisation techniques.

Most software synthesis tools make a distinction between software synthesis and software compilation. In fact most methods perform a two stage process where the synthesis process outputs a program that is compiled using an ordinary optimising compiler. The hope is that the compilers local optimisation is able to improve the size and speed of the code.

Commercial tools

Several commercial tools for software design and simulation offer code generation capabilities. We mention a few of them as examples of tools in different areas and with different purposes. A distinguishing characteristic lies in if the generated code is considered to be production quality or mostly intended for prototypes.

From the first category we mention Telelogics TAU [Tel03] which is a software development tool based on the SDL language [ITU02] (that is based on extended finite state machines) that can generate C, C++ and CHILL code. The tool is mainly used in telecommunication systems.

IAR visualSTATE [IAR03] is a graphical design tool for developing embedded systems software. The tool uses finite state machines for system specification and generates compact C/C++ code. VisualSTATE is targeted at small embedded systems with close connection to the hardware.

An example from the second category of code-generators for prototypes is the Simulink toolbox for MATLAB for which there exist several code generation tools. Most of them are targeted at generating code for prototypes or real-time validation of the product. With these code generators it is possible to generate code for discrete time, continuous time and hybrid system.

Few of the commercial tools have a theoretical underpinning. Not even those that are considered to produce production quality code. This means that they can not provide guarantees that the generated code has, for example, a specified behaviour or a desired real-times property.

Synchronous languages

For the family of so called synchronous languages there exist efficient code generation techniques. Especially we mention Esterel [Ber] as an example of a synchronous language for reactive systems. The usage of the term synchronous in this context means that the reaction to input is assumed to take neglectible time. The program waits for input and reacts with an output without any time delay. This abstraction simplifies the implementation, verification and optimisation of the program.

The semantic interpretation of Esterel developed in [Ber99] translates an Esterel program into a Boolean circuit, which is very similar to an electrical circuit. This implies that the program is easy to compile into a hardware implementation. The software implementation of the program is also based on the same circuit interpretation with a loop that reads the input and reaction that interpret the generated circuit. The similarity between the semantic interpretation and a hardware implementation makes it natural that Esterel has been used for hardware/software co-design.

1.4 Contributions

The main contributions of this thesis include a deterministic semantics for the TAT language and an implementation of a code generator for embedded real-time systems based on the deterministic semantics. In contrast to most code generators in commercial tools the code generated by the presented generator is constructed to preserve the safety properties of a design model.

The TAT language was originally designed to describe general scheduling problems with aperiodic task arrival patterns. In this thesis we have extended its use to a design language for real-time embedded systems.

In more detail, the contributions of this thesis are:

- A code generator of optimised executable code from analysed designs. We give a formal definition of the design language TAT and its operational semantics. In this thesis we extend the notion of tasks with data variables shared between the automata structure and the task code and formalise the interface between them. We also present how TAT-design can be analysed with respect to schedulability, bounded resource requirements and other safety properties.

- A deterministic semantics of TAT-designs that define an implementable subset of the operational semantics. The deterministic semantics define a subset of the behaviours of the operational semantics and hence safety properties are preserved. Based on the deterministic semantics we show how to generate executable code for embedded real-time systems and present a prototype implementation for brickOS. The code generation has been integrated in the tool Times.

- A specialised interpretation of the task notion in TAT where the tasks are systems of differential equations. In this interpretation a TAT-designs defines hybrid automata. The hybrid automata are used to model the environment of a control design and we present an animator that can be used to graphically illustrate the behaviour of the controller in a simulated environment.

- An application of the TAT language as a design language is presented in a case-study where a controller for an industrial production cell has been developed. The design of the controller is presented in detail as well as the results of the analysis and automatically generated code.

1.5 Outline

This thesis is organised as follows. Chapter 2 contains a presentation of the TAT language, with syntax, semantics and analysis. Chapter 3 describes code generation from TAT models, based on a deterministic semantics, and a prototype implementation. Chapter 4 presents an animator for hybrid automata with implementation and some examples. Chapter 5 describes a case study where the techniques presented in this thesis have been applied. In conclusion we review the future work and open questions in Chapter 6.

Chapter 2

TAT: a Language for Design of Embedded Systems

In this chapter we present a design language that will be used throughout this thesis. The language is named TAT (timed automata with tasks). The language is based on Alur and Dill's Timed Automata [AD94], with extensions to make it suitable for practical development of embedded systems. Essentially the extensions are:

- executable tasks associated with actions,
- parallel composition into networks of automata and
- data variables.

Parallel composition and data variables are rather standard extensions to timed automata. They can for example be found in the input languages to model checking tools such as Uppaal [ABB+01]. A newer extension to timed automaton is that of *tasks*. Informally tasks are executable pieces of code that are associated with edges of the automata. When the edge is taken the associated task is triggered.

The notion of tasks in timed automata is introduced in [EWY99] and [FPY02] as a general model for describing real-time scheduling problems. Each task is associated with two parameters, a deadline and a worst case execution time. When a task is triggered it is scheduled for execution, and the problem to solve is if the system may evolve to a situation where any of the tasks cannot meet its deadline. We will discuss the solution of this problem for our setting in Section 2.4.

In this thesis we will use the same model as in the works cited above, but we are also interested in what the code in the tasks actually does. Roughly speaking we use timed automata to describe the control flow of a program and let tasks contain code that accesses the hardware and performs computations. We assume a task to be a reliable and well behaved software component with known properties such as memory utilisation and computation time. A major addition to the task model in the cited works is that we allow tasks to update data variables of the timed automata and thus allow tasks to influence the control flow of the model.

2.1 Syntax of TAT

The basic structure of a TAT design model is a finite state automaton extended with clocks, data variables and tasks. We introduce the syntax with an example.

Example 2.1 *A simple timed automaton with a task is shown in Figure 2.1*[1]*. It has two* locations*, two* edges *and one* clock *(the label* x*). Initially the automaton is in location* Start *(indicated by the arrow with a dot). To ensure progress the initial location is labelled with an* invariant *($x <= 3$). An automaton may only stay in a location while the invariant is* true*, so the initial location must be left within 3 time units. The* guard *($x >= 3$) on the edge from* Start *to* Loop *is combined with the invariant to specify that the discrete transition to* Loop *is taken after exactly three time units. The location* Loop *is labelled with the task* TaskA *which is released when the location is entered. In* Loop *there is no invariant, so the automaton may remain there forever. It also has the possibility to non-deterministically take the loop edge when x is greater than 2 to reset x to zero.*

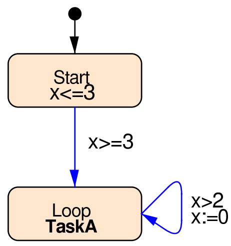


Figure 2.1: A timed automaton with a task.

Syntactically a TAT model consists of *clock variables*, *data variables*, *actions*, *tasks* and *automata structure*.

Clock variables are used to measure time progress with uniform speed. We will use $\mathcal{C}$ to identify a finite set of real-valued clock variables ranged over by x, y, z etc. Clocks may be tested using guards and invariants, and reset on action transitions. We identify clocks by their names which are strings beginning with a character.

Data variables are assumed to take values from finite data domains, $\mathbb{D} \subset \mathbb{Z}$. We will use $\mathcal{D}$ to identify a finite set of data variables ranged over by i, j, k etc. Data variables can be tested using guards, and updated on action transitions and by tasks. Data variables are identified by their names which are strings beginning with a character. Variables may be initialised with any value from their domain.

Actions represent interaction between automata. Transitions may be labelled with action labels of the form $action\_name$ $\{!|?\}$, where $action\_name$ is a string beginning with a character, and ! indicates a sending action and

[1] All figures showing automata are drawn using the tool Times `www.timestool.com`

? indicates a receiving one. Transitions may also be labelled with an internal (invisible) action indicated by τ. We will use $\mathcal{A}ct = \{\alpha! | \alpha \in \mathcal{A}\} \cup \{\alpha? | \alpha \in \mathcal{A}\} \cup \{\tau\}$ to denote the set of actions, where $\mathcal{A}$ is a finite set of action names ranged over by α, β etc.

Tasks are executable pieces of code. We will use $\mathcal{P}$ ranged over by P, Q, etc. to denote a finite set of task types. Tasks are associated with action labels and are released when the action is taken.

Guards are boolean expressions over clocks and data variables, used to determine if an edge is enabled. We will use $\mathcal{B}$, ranged over by g to denote the set of guards. The syntax of guards is defined by a grammar, where *NAT* denotes the natural numbers, *clid* denotes clock names and *varid* denotes variable names. A list of guards is interpreted as a conjunction of the listed guards.

$$
\begin{array}{rcl}
guard\_list & \rightarrow & guard\ (',' \ guard)^* \\
guard & \rightarrow & cguard \mid iguard \\
cguard & \rightarrow & clid\ rel\ cexpr \mid clid\ rel\ clid \mid clid\ rel\ clid + cexpr \\
rel & \rightarrow & '<' \mid '<=' \mid '>=' \mid '>' \mid '==' \\
iguard & \rightarrow & iexpr\ rel\ iexpr \mid iexpr != iexpr \\
iexpr & \rightarrow & varid \mid varid\ '['\ iexpr\ ']' \mid NAT \mid '-'\ iexpr \mid '('\ iexpr\ ')' \mid \\
 & & iexpr\ op\ iexpr \mid '('\ iguard\ '?'\ iexpr\ ':'\ iexpr\ ')' \\
cexpr & \rightarrow & NAT \mid varid \mid '('\ cexpr\ ')' \mid cexpr\ op\ cexpr \mid '-'\ cexpr \\
op & \rightarrow & '+' \mid '-' \mid '*' \mid '/'
\end{array}
$$

Assignments are updates of clocks and data variables on transitions. We shall use R, ranged over by r etc, to denote the set of assignments and we call any finite sequence of assignments an update and let $\mathcal{U}$, ranged over by u etc, denote the set of such sequences. The syntax for assignments is defined by the grammar for $assign\_list$:

$$
\begin{array}{rcl}
assign\_list & \rightarrow & assign\ (',' \ assign)^* \\
assign & \rightarrow & iassign \mid cassign \\
cassign & \rightarrow & clid\ ':='\ cexpr \\
iassign & \rightarrow & varid\ ':='\ iexpr
\end{array}
$$

where *clid*, *varid*, *cexpr* and *iexpr* are the same as defined for guards above.

Invariants are boolean expressions over clocks. We will use $\mathcal{J}$ to denote the set of invariants. A list of invariants is interpreted as the conjunction of the listed invariants. The syntax of invariants is defined by the grammar:

$$
\begin{array}{rcl}
invariant\_list & \rightarrow & inv\ (',' \ inv)^* \\
inv & \rightarrow & clid\ ('<' \mid '<=')\ cexpr
\end{array}
$$

Definition 2.1 *(Syntax) A Timed Automaton with Tasks over actions $\mathcal{A}ct$, clocks $\mathcal{C}$, data variables $\mathcal{D}$ and tasks $\mathcal{P}$ is a tuple $\langle N, l_0, E, I, M \rangle$ where*

- *N is a finite set of locations ranged over by l, m, n,*
- *$l_0 \in N$ is the initial location,*
- *$E \subseteq N \times \mathcal{B} \times \mathcal{A}ct \times \mathcal{U} \times N$ is the set of edges,*
- *$I : N \mapsto \mathcal{J}$ is a function assigning each location with an invariant, and*
- *$M : \mathcal{A}ct \hookrightarrow 2^{\mathcal{P}}$ is a partial function assigning actions with sets of tasks.*[2]

Example 2.2 *In Figure 2.2 we show an automaton with associated tasks, P, Q and R. Tasks are shown as labels in bold font in the locations, this is a shorthand for having*

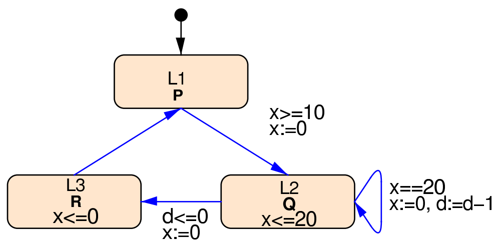


Figure 2.2: A timed automaton with tasks.

an action label with the task associated to all incoming edges. We will use this notation throughout the thesis. In location L2 *task Q is released periodically with period 20 until the data variable d becomes less than 0.*

Figure 2.3 shows one possible schedule for the tasks when d is initially 0.

[2]Note that M is a partial function meaning that some actions have no task.

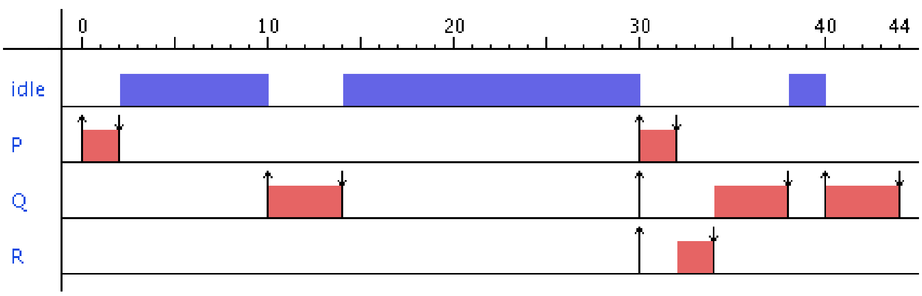


Figure 2.3: A Gantt chart showing a task schedule of the system shown in Figure 2.2.

2.1.1 Tasks

A task is an executable piece of code that is triggered by action transitions. A task type may have different instances that are copies of the same program with different inputs. Each task type in $\mathcal{P}$ has known parameters such as execution time, denoted by $C(P)$, and relative deadline, denoted by $D(P)$.

We will use tasks to encapsulate computations and/or interact with the environment of the controller. The external behaviour of a task is determined by its *interface*. A task interface has two components; a set of data variables that the task uses, and a list of assignments to shared data variables. We will use $V(P) \subseteq \mathcal{D}$ to denote the set of variables used by task P and $A(P)$ to stand for the assignments of task P.

Before the task starts to execute it takes a copy of the data variable values it needs and stores them in a local context. The task will use the local copies for its computations and update the global data variables according to the assignments when it finishes.

There is an interesting design choice concerning the time point when the used variables are copied to the local context. There are two reasonable alternatives, the local context can be copied either at task release or when the task starts to execute. Note that a task may start executing much later than its release when there are tasks released with higher priorities. The first alternative corresponds to a task that acts as a sub-procedure of the automaton. The local copies of the variables corresponds to parameters to the procedure. The second alternative is more like a data flow program where one task produces values needed by the subsequent task.

Formally a task assignment has the same form as updates on edges and the grammar is the same as for *assign_list* defined above. Task updates are evaluated on the variables in the tasks local context but updates the global data variables. Note that the variables that the task updates doesn't have to be the same as those it uses.

We can observe that computations in a TAT design model are of two kinds. Assignments on edges are synchronous with the execution of the automaton while assignments performed by the tasks are asynchronous.

In general the number of task parameters is not fixed but depends on the type of analysis we wish to perform on the design. Other parameters could be resource requirements such as memory consumption or shared resource protection by semaphores. In this thesis we will handle the tasks with interfaces and fixed priorities.

2.2 Semantics of TAT

We will consider three different semantic interpretations of TAT. In the first interpretation we extend the ordinary operational semantics of standard timed automata with a task queue. In the following chapters we will also consider a deterministic interpretation of the operational semantics as a basis for code generation and a discrete semantics for hybrid automata. The different semantics are intended for different stages of the development process and the relation between them will be described later.

Semantically a timed automata extended with tasks is a labelled transition system. A semantic state is a triple (l, σ, q) where l is the current location, σ denote the current values of the variables and q is the current task queue.

Variable values are denoted by *valuations*. A valuation is a function that maps clock variables to the non-negative reals and data variables to their data domain. We use $\mathcal{V}$, ranged over by σ, to denote the set of valuations. We will also use ν to denote *local* valuations, or context, which maps the data variables used by a task to their data domains, $\nu : V(P) \mapsto \mathbb{D}$.

To update the variable values as the effect of delays we write $\sigma{+}t$, where $t \in \mathbb{R}_{\geq 0}$. This denotes the valuation which updates each clock $x \in \mathcal{C}$ with $\sigma(x) + t$. To update the variable values as the effect of an assignment we write $r_\nu[\sigma]$ which is the valuation that maps each variable χ to the value of $\mathcal{E}$ evaluated in ν if $(\chi := \mathcal{E}) \in r$. We shall use $\sigma_{V(P)}$ to denote the valuation which restricts σ to the variables $V(P)$ used by task P. When an assignment is evaluated on the same valuation as it updates we will sometimes skip the subscript and write $r[\sigma]$.

For an update $u \in \mathcal{U}$ consisting of assignments $r, r', \ldots, r''$ we will use $u_\nu[\sigma]$ to denote the consecutive application of each assignment in the sequence, that is $r''_\nu\,[\ldots[r'_\nu\,[r_\nu[\sigma]]]]$.

The task queue is represented in the form $[P_1\,(c_1, d_1, \nu_1)\,, P_2\,(c_2, d_2, \nu_2)\,, \ldots, P_n\,(c_n, d_n, \nu_n)]$, where $P_i\,(c_i, d_i, \nu_i)$ denotes a released instance of task type P_i with remaining computation time c_i, relative deadline d_i and context ν_i.

We will assume that the released tasks are executed on a single processor system according to a scheduling strategy Sch. The strategy determines the order of execution of the released tasks and could for example be fixed priority scheduling (FPS) or earliest deadline first (EDF). In general we assume that the scheduling strategy is a sorting function which orders the queue so that the task to execute is at the head of the queue.

Transitions are of two types, discrete action transitions where control locations changes and delay transitions where time passes. Informally a discrete transition corresponds to the release of a task and a delay transition to the execution of the task at the front of the task queue. A discrete transition will result in a re-sorting of the queue where the newly released tasks (if any) are inserted into the queue. A delay transition of t time units will result in the execution of a task in the queue for t time units (decreasing its remaining computation time with t). If the remaining computation time becomes zero the task should be removed from the queue. We will use two functions to handle the queue:

- Sch is a sorting function for the task queue. It may only change the order of the elements in the queue (not their values). For example, $\mathsf{EDF}\,([P(3.1, 10, \nu_P), Q(4, 5.3, \nu_Q)]) = [Q(4, 5.3, \nu_Q), P(3.1, 10, \nu_P)]$. We call such sorting functions scheduling strategies and allow them to be either preemptive or non-preemptive.

- Run is a function which given a queue q and a real value t will return the state of the queue and the value of the variables after t time units execution of the tasks at the head of the queue. For example let $q = [Q(4, 5.3, x = 4), P(3.1, 10, x = 4)]$, $\sigma = \{x = 4\}$ and the assignments of task Q be $A(Q) = \{x := x/2\}$. Then $\mathsf{Run}\,(q, 6) = [P(2.1, 4, x = 4)]$ and $\sigma = \{x = 2\}$ in which the first task has finished its execution and updated the valuation and the second task has been executed for 2 time units.

Definition 2.2 *(Operational semantics) Given a scheduling strategy* Sch *the semantics of a timed automata with tasks* $\langle N, l_0, E, I, M\rangle$ *with initial state* (l_0, σ_0, q_0) *is a transition system defined by the following rules:*

- $(l, \sigma, q) \xrightarrow{\alpha}_{\mathsf{Sch}} (l', u[\sigma], \mathsf{Sch}(M(\alpha) :: q)$ *if* $l \overset{g,\alpha,u}{\longrightarrow} l'$ *and* $\sigma \models g$
- $(l, \sigma, q) \xrightarrow{t}_{\mathsf{Sch}} (l, \sigma + t, \mathsf{Run}\,(q, t))$ *if* $(\sigma + t) \models I(l)$ *and* $C\,(\mathsf{Hd}\,(q)) > t$
- $(l, \sigma, q) \xrightarrow{t}_{\mathsf{Sch}} (l, (A\,(\mathsf{Hd}\,(q))\,[\sigma]) + t, \mathsf{Run}(q, t))$ *if* $(\sigma + t) \models I(l)$ *and* $C\,(\mathsf{Hd}\,(q)) = t$

where $M(\alpha) :: q$ *denotes the queue with a new instance inserted in q for each task type in* $M(\alpha)$, $\mathsf{Hd}\,(q)$ *denotes the first element of q and* $\sigma \models g$ *denote that the valuation* σ *satisfies the guard* g.[3]

The first rule defines that an action transition can be taken whenever there exist an enabled edge i.e. an edge with a satisfied guard. When the transition is taken, the variables are updated according to the updates of the edge, and the tasks associated with the action (if any) are inserted into the task queue using the function Sch.

There are two kinds of delay transitions as defined by the second and third rules. In the second rule, the delay transition corresponds to that the first task in the queue is executed. The transition is enabled when the invariant is still satisfied after the delay and the task will not complete its execution during the delay. The transition results in updates of the clock variables and a decrease of remaining execution time and deadline of the task at the head of the queue.

In the third rule, the delay transition corresponds to completing the execution of the task at the head of the queue. The delay is equal to the remaining execution time of the task. The transition results in updates of the clock variables according to the delay and the data variables according to the assignments of the task. Note that we assume fixed execution times for tasks. It is possible to extend the model and the analysis so that the execution times vary between a best and a worst case.

In the following, when there is no risk of confusion, we shall omit the subscript Sch from the transition relation $\longrightarrow_{\mathsf{Sch}}$.

Example 2.3 *We illustrate the semantic rules with the help of the automaton in Figure 2.2. The tasks have parameters as in Table 2.1.*

Table 2.1: Task parameters for automaton in Figure 2.2.

task type	computation	deadline	assignment	uses
P	2	4	$d := (d < 0?0 : 1)$	d
Q	4	20		
R	2	7	$d := (d < 0?0 : 5)$	d

[3]In case the task queue is empty we interpret the conditions $C\,(\mathsf{Hd}\,(q)) > t$ and $C\,(\mathsf{Hd}\,(q)) = t$ as true which means that the two last transition rules become equal and correspond to a delay transition as in ordinary timed automata.

*Assume that earliest deadline first (*EDF*) is adopted to schedule the task queue. The automaton is initially in the state* $(L1, [x = 0, d = 0], [P(2, 4, d = 0)])$*. One possible run of the automaton is the following sequence of transitions:*

$$
\begin{aligned}
&(L1, [x = 0, d = 0], [P(2, 4, d = 0)]) \\
&\xrightarrow{1.5} (L1, [x = 1.5, d = 0], [P(0.5, 2.5, d = 0)]) \\
&\xrightarrow{9} (L1, [x = 10.5, d = 1], []) \\
&\xrightarrow{\tau} (L2, [x = 0, d = 1], [Q(4, 20, \varnothing)]) \\
&\xrightarrow{4.3} (L2, [x = 4.3, d = 1], []) \\
&\xrightarrow{15.7} (L2, [x = 20, d = 1], []) \\
&\xrightarrow{\tau} (L2, [x = 0, d = 0], [Q(4, 20, \varnothing)]) \\
&\xrightarrow{2.3} (L2, [x = 2.3, d = 0], [Q(1.7, 17.7, \varnothing)]) \\
&\xrightarrow{\tau} (L3, [x = 0, d = 0], [R(2, 7, d = 0), Q(1.7, 17.7, \varnothing)]) \\
&\xrightarrow{\tau} (L1, [x = 0, d = 0], [P(2, 4, d = 0), R(2, 7, d = 0), Q(1.7, 17.7, \varnothing)])
\end{aligned}
$$

In this run no task violates its deadline. In fact this automaton is always schedulable if EDF *is used as scheduling strategy.*

2.3 Networks of timed automata

A design in TAT can be composed of several components where each component is a TAT model. We call such a composition of automata a *network of timed automata with tasks*, or network of automata in short. The components interact through synchronisation actions and global variables. We introduce a CCS-like[Mil89] parallel composition operator $|$. We use $\bar{C}$ to denote a network of automata, i.e. the parallel composition $C_1|...|C_n$.

As in CCS parallel components may synchronise with each other using complementary actions from $\mathcal{A}ct$. We will call a set of complementary actions a *channel* where *channel_name*! is the complement of *channel_name*?. If one component has an edge labelled with an action and another component has an enabled edge labelled with the complementary action they may perform a *compound* transition. We will make a distinction between the actions and call the action with an exclamation mark the *sending* action and the one with the question mark the *receiving* action. The edge with the sending label will perform its updates before the edge with the receiving label. We will sometimes also use $\bar{\alpha}$ to denote the complementary action of α and let $\alpha = \bar{\bar{\alpha}}$.

We will make a distinction between internal and external actions. Let $\mathcal{A}ct_{\mathcal{I}}$ and $\mathcal{A}ct_{\mathcal{O}}$ be disjoint subsets of $\mathcal{A}ct$. We let $\mathcal{A}ct_I$ contain the "internal" actions which are used to synchronise between the automata within the network and we let $\mathcal{A}ct_O$ contain the "external" actions which are used to synchronise with the environment.

We will assume that the network $\bar{C}$ is restricted by the actions in $\mathcal{A}ct_{\mathcal{I}}$, that is (using CCS notation) $\bar{C} = (C_1|\cdots|C_n) \setminus \mathcal{A}ct_{\mathcal{I}}$. Thus some of the actions in

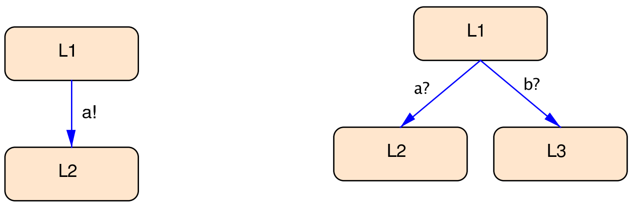
$\mathcal{A}ct$ are restricted to synchronisation within the network and some are used to interact with the environment.

We use a *location vector* $\bar{l}$ to denote the configuration of locations in a network $\bar{C}$. In the vector the ith element l_i is a location of the ith component C_i in the network. We will write $\bar{l}\,[l'_i/l_i]$ to denote the location vector where the ith element l_i of $\bar{l}$ is replaced by l'_i.

In the obvious way the set of clocks in a network is the union of the clocks of each component, $\mathcal{C} = \bigcup_{1\leq i\leq n} \mathcal{C}_i$. And similarly for the data variables $\mathcal{D} = \bigcup_{1\leq i\leq n} \mathcal{D}_i$ and the set of actions $\mathcal{A}ct = \bigcup_{1\leq i\leq n} \mathcal{A}ct_i$.

In a network of timed automata with tasks there is one unified queue and we still assume only one processing unit that can execute tasks.

Example 2.4 *An example of a network containing the components C_1 and C_2 is presented in Figure 2.4. The set of actions is $\mathcal{A}ct = \{a!, a?, b!, b?\}$ consisting of internal actions $\mathcal{A}ct_{\mathcal{I}} = \{a!, a?\}$ and external actions $\mathcal{A}ct_{\mathcal{O}} = \{b!, b?\}$. The initial configuration is represented by the location vector $\langle L1, L1\rangle$. When executed two things may happen, either the network perform the compound transition on the internal action a to configuration $\langle L2, L2\rangle$, or the environment synchronise using the external action b transferring the network to configuration $\langle L1, L3\rangle$.*



(a) Component C_1 (b) Component C_2

Figure 2.4: Synchronising components.

2.3.1 Semantics of networks of automata

The composition of timed automata with tasks into a network will impose restrictions on the transitions the network can perform compared to the separate components. Let $\bar{C}$ be a network. We have a rule defining delay transitions:

$$\frac{\forall i : (l_i, \sigma, q) \xrightarrow{t} (l_i, \sigma + t, q')}{(\bar{l}, \sigma, q) \xrightarrow{t} (\bar{l}, \sigma + t, q')} \tag{2.1}$$

That is, the network may delay for t time units if and only if all the components may separately delay for t time units.

We have a rule defining synchronisation between component automata:

$$\frac{(l_i, \sigma, q) \xrightarrow{\alpha?} (l'_i, u_i[\sigma], q') \qquad (l_j, \sigma, q) \xrightarrow{\alpha!} (l'_j, u_j[\sigma], q'')}{(\bar{l}, \sigma, q) \xrightarrow{\tau} \left(\bar{l}\,[l'_i/l_i]\,\left[l'_j/l_j\right], u_i\left[u_j\left[\sigma\right]\right], q'''\right)} \quad i \neq j \tag{2.2}$$

That is, two components may synchronise if and only if they have enabled transitions with complementary labelled edges. The resulting queue includes the tasks associated with both action labels. Note that updates of variables are ordered so that the component with the sending (!) action will perform them before the component with the receiving (?) action.

2.3.2 Environments

We will consider networks of automata to act as controllers of objects in an environment. These external objects may be real physical objects such as motors or part of a simulated environment. We will view the environment as objects that can *consume* actions produced by the controller and independently *produce* actions consumed by the controller. The environments ability to produce and consume actions can change over time or after having produced or consumed an action.

We shall define environments as timed labelled transition systems $\mathsf{E} = \left\langle \mathsf{Env}, \left\{ \stackrel{t}{\Rightarrow} : t \in T \right\} \right\rangle$, where Env is the set of states ranged over by E, T is a set of transition labels and $\stackrel{t}{\Rightarrow} \subseteq \mathsf{Env} \times \mathsf{Env}$ is a transition relation. We take T to be $\mathcal{A}ct_{\mathcal{E}} \cup \mathbb{R}^{\geq 0}$, the union of actions and delays. Where $\mathcal{A}ct_{\mathcal{E}}$ is the set of action names. Let d range over the values of delays.

We shall understand the transition $E \stackrel{\alpha}{\Rightarrow} E'$ as that the environment performs action α and evolves to E' and the transition $E \stackrel{u}{\Rightarrow} E'$ as that the environment idles for u time units and evolves to E', where possibly $E = E'$. We say that an action α is *enabled* in state E iff $\exists \alpha, E' : E \stackrel{\alpha}{\Rightarrow} E'$ and we use $E \stackrel{\alpha}{\Rightarrow}$ to denote that α is enabled in E.

Let $\mathsf{E}\left[\bar{C}\right]$ denote that the network $\bar{C}$ is executed within the environment E. Then the set of action names in the environment is restricted to $\mathcal{A}ct_{\mathcal{E}} = \mathcal{A}ct_{\mathcal{E}} \cap \mathcal{A}ct_{\mathcal{O}}$. The controller is only constructed to react on the actions in $\mathcal{A}ct_{\mathcal{O}}$ and whatever the the environment is able to do otherwise cannot be observed by the controller. With this restriction environments and networks may interact with each other using the following rules. For synchronisations,

$$\frac{\left(\bar{l}, \sigma, q\right) \xrightarrow{\alpha} \left(\bar{l}', \sigma', q'\right) \qquad \mathsf{E} \stackrel{\bar{\alpha}}{\Rightarrow} \mathsf{E}'}{\mathsf{E}\left[\left(\bar{l}, \sigma, q\right)\right] \stackrel{\tau}{\Rightarrow} \mathsf{E}'\left[\left(\bar{l}', \sigma', q'\right)\right]} \tag{2.3}$$

and for delays,

$$\frac{\left(\bar{l}, \sigma, q\right) \xrightarrow{t} \left(\bar{l}, \sigma + t, q'\right) \qquad \mathsf{E} \stackrel{t}{\Rightarrow} \mathsf{E}'}{\mathsf{E}\left[\left(\bar{l}, \sigma, q\right)\right] \stackrel{t}{\Rightarrow} \mathsf{E}'\left[\left(\bar{l}, \sigma + t, q'\right)\right]} \tag{2.4}$$

An environment may be described using any appropriate formalism that can be defined as a labelled transition system interacting using the rules above. In this thesis we will use both timed automata (in the next Section) and hybrid automata (in Chapter 4) to describe the environment of a controller.

2.4 Analysis

In this section we will discuss how to analyse a TAT design. It has been shown that reachability and schedulability are decidable problems for the TAT model see [EWY99] and [FPY02].

The goal of the analysis is to establish if a control design, given as a network of automata $\bar{C}_{Design} = C_1|C_2|\dots|C_n$, when executed in an environment E, have desired properties. The properties we are interested in are those that are required for the design to be implementable. We are also interested in safety properties and properties related to possible optimisations of code generation.

In this section we assume that the environment is described using timed automata *without* tasks. The models of the objects in the environment are executed in parallel with the controller forming a combined network of automata, $\bar{C}_{System} = \bar{C}_{Env}|\bar{C}_{Design}$.

Informally reachability is to determine if there is a sequence of transitions allowed by the system network leading to a specific state.

Definition 2.3 *(Reachability) We use $(\bar{l}, \sigma, q) \longrightarrow (\bar{l}', \sigma', q')$ to stand for both action transitions $(\bar{l}, \sigma, q) \xrightarrow{a} (\bar{l}', \sigma', q')$ and delay transitions $(\bar{l}, \sigma, q) \xrightarrow{t} (\bar{l}, \sigma', q')$. Then a state $(\bar{l}, \sigma, q)$ is reachable from an initial state $(\bar{l}_0, \sigma_0, q_0)$ iff $(\bar{l}_0, \sigma_0, q_0) \longrightarrow^* (\bar{l}', \sigma', q')$. Where $(\bar{l}, \sigma, q) \longrightarrow^* (\bar{l}', \sigma', q')$ denotes a sequence of zero or more transitions $(\bar{l}_0, \sigma_0, q_0) \longrightarrow (\bar{l}_1, \sigma_1, q_1) \longrightarrow \cdots \longrightarrow (\bar{l}_n, \sigma_n, q_n)$.*

We will use reachability analysis to check for *safety properties* that is, that a certain undesired situation is guaranteed never to be reached. We will also use it to check for *invariant properties* that is, that a property is guaranteed to hold always. Properties are represented as sets of states using temporal logic formulae.

To determine reachability of a state of timed automata with tasks the queue must be considered. But since, in general, a task queue may be unbounded we must determine if a specific design has a bounded queue. Formally we define boundedness of the task queue in the following way:

Definition 2.4 *(Boundedness) A timed automaton with tasks A with initial location $(\bar{l}_0, \sigma_0, q_0)$ is bounded by the queue length* n *iff for all reachable states $(\bar{l}, \sigma, q)$, $|q| \leq n$, where $|q|$ is the number of task instances in the queue.*

Note that checking for the boundedness property in a timed automata with tasks with a given queue length is an invariant property i.e. an instance of reachability. We may also find the least upper bound of the queue length by constructing all reachable states and taking the maximal value of $|q|$.

The schedulability problem, that is to check if all tasks always meet their deadlines, is central to implementability of a TAT design.

Definition 2.5 *(Schedulability) A state $(\bar{l}, \sigma, q)$ with $q = [P_1(c_1, d_1, \nu_1), \dots, P_n(c_n, d_n, \nu_n)]$ is a* failure state, *denoted $(\bar{l}, \sigma, \mathsf{Error})$, if there exist an i, $i \in [1..n]$, such that $c_i \geq 0$ and $d_i < 0$, that is, a task failed in meeting its deadline. An automaton A is non-schedulable with scheduling policy* Sch *iff $(\bar{l}_0, \sigma_0, q_0) \longrightarrow^*_{\mathsf{Sch}} (\bar{l}, \sigma, \mathsf{Error})$ for some $\bar{l}$ and σ. Otherwise, we say that A is schedulable with* Sch*.*

We may check if a state (l, σ, q) is a *failure state* with a standard test. We say that (l, σ, q) is *schedulable* with Sch if $\mathsf{Sch}(q) = [P_1(c_1, d_1, \nu_1), \ldots, P_n(c_n, d_n, \nu_n)]$ and $\left(\sum_{i \leq k} c_i\right) \leq d_k$ for all $k \leq n$, otherwise it is a failure state. Alternatively, we say that a network of timed automata with tasks is schedulable with Sch if all its reachable states are schedulable with Sch.

Note that all schedulable states are also bounded. Consider the maximal number of instances of a task type in a schedulable queue. This number is bounded by $\lceil D(P)/C(P) \rceil$ since a task instance that has started cannot be preempted by another instance of the same task type. The size of all schedulable queues is bound by the sum for all task types, i.e. $\sum_{P \in \mathcal{P}} \lceil D(P)/C(P) \rceil$.

Schedulability for timed automata with tasks has been studied in for example [EWY99, FPY02] where it has been shown that the problem is decidable for both non-preemptive and preemptive scheduling policies.

The basic idea in these works is to encode the schedulability problem as a reachability problem of ordinary timed automata. This allows the application of some of the efficient reachability algorithms for timed automata that are implemented in for example the model-checker Uppaal[BLL+96].

A very efficient encoding of the schedulability checking problem for timed automata with tasks is presented in [FMPY03]. There it is shown that for a model where the tasks do not update shared data variables and with a fixed priority scheduling strategy the problem can be solved using reachability analysis using only two extra clocks. For models with tasks that update shared data variables and with a fixed priority scheduling strategy it is shown that $n + 1$ extra clocks are needed, where n is the number of task types in the model.

Chapter 3

Code Synthesis

Code synthesis is to generate executable code from the design model so that the generated code implements the behaviour of the model. We assume that the generated code is to be executed on a target platform that guarantees the synchronous hypothesis and that the tasks consume their given execution times.

3.1 Deterministic semantics of TAT

In general the behaviour of the design model according to the operational semantics in Definition 2.2 is non-deterministic. While this simplifies analysis by allowing abstract non-deterministic descriptions of objects in the environment it introduces problem for code-generation. It is very hard to implement a non-deterministic behaviour on a deterministic device such as a CPU, instead we adopt a deterministic semantics for TAT as the basis for the code generation. The deterministic semantics defines a subset of the behaviour of the controller so that safety properties (including schedulability) are preserved.

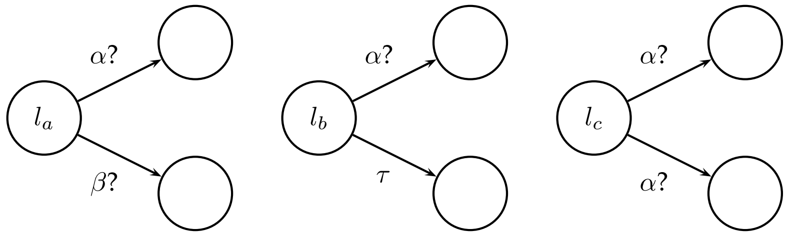
3.1.1 Resolving non-determinism

The sources of non-determinism in the operational semantics are time-delays and external actions. We use *time non-determinism* to mean that an enabled transition can be taken at any time point within the time-zone while we use *external non-determinism* to mean that several actions may be simultaneously present from the environment which results in several enabled transitions in a state.

We refine the design model for the controller as follows:

- *External non-determinism* is resolved by defining priorities for action transitions in the controller. The enabled transition with the highest priority will always be taken.

- *Time non-determinism* is resolved by adopting the so-called maximal-progress assumption [Yi91]. Maximal-progress means that the controller should take all enabled transitions until the system stabilises, i.e. no more action transitions are enabled. Similar ideas have been adopted in the



(a) (b) (c)

Figure 3.1: Situations with external non-determinism

> *asynchronous time model* in the Statemate semantics of Statecharts [HN96] and the *run-to-completion* of UML statecharts [LP99].

We assign priorities to action transitions so that an edge in a given automaton in the network has a static and unique priority. The components of the network is ordered so that lower priority components may only take an action when the higher priority components are unable to do so. Within each component the priority of an action is determined by the priority of the action label of the edge. For locations which have several outgoing edges with the same action label we define a local priority order among those edges.

Let $\mathsf{Pr}(l_i \xrightarrow{g,\alpha,u} l_i')$ be a function that assigns unique priorities to edges in component i labelled with action α. The priority of a given edge is computed based on the priorities for components, actions and outgoing edges.

We illustrate the priority assignment within a component by considering the three automata in Figure 3.1. In (a) the environment may synchronise on both α and β, the priority of action labels determines which transition to take. In (b) the the external action (α) has priority over the internal (τ) action, interaction with the environment has priority. In other words τ actions have lower priority than all other actions. In (c) the choice is resolved by the local priority order on outgoing edges from the location l_c.

3.1.2 Deterministic semantics

We have now a refined deterministic version of the operational semantics for timed automata with tasks:

Definition 3.1 *(Deterministic semantics) Let $\bar{A} = \langle N, \bar{l}_0, E, I, M \rangle$ be a network of automata, and* E *an environment in which it executes. Given a scheduling strategy* Sch *and a function* Pr *that assigns unique priorities to edges the deterministic semantics is a labelled transition system defined by these rules:*

$$\frac{(l_i, \sigma, q) \xrightarrow{\alpha} (l_i', \sigma', q') \qquad \mathsf{E} \overset{\bar{\alpha}}{\Rightarrow} \mathsf{E}'}{\mathsf{E}\left[(\bar{l}, \sigma, q)\right] \overset{\tau}{\Rightarrow} \mathsf{E}'\left[(\bar{l}\,[l_i'/l_i], \sigma', q')\right]} \; \mathsf{Pr}\left(l_i \xrightarrow{g,\alpha,u} l_i'\right) > \mathsf{Pr}\left(l_j \xrightarrow{g',\alpha',u'} l_j'\right)$$

for all edges $l_j \xrightarrow{g',\alpha',u'} l_j'$ *such that* $\sigma \models g'$ *and* $\mathsf{E} \overset{\bar{\alpha}'}{\Rightarrow}$

$$\frac{(\bar{l},\sigma,q) \xrightarrow{t} (\bar{l},\sigma+t,q') \qquad \mathsf{E} \overset{t}{\Rightarrow} \mathsf{E}'}{\mathsf{E}\left[(\bar{l},\sigma,q)\right] \overset{t}{\Rightarrow} \mathsf{E}'\left[(\bar{l},\sigma+t,q')\right]}\ \mathsf{E}'\left[(\bar{l},\sigma+t',q)\right] \overset{\overline{\tau}}{\not\Rightarrow} \quad \forall t' < t$$

The first rule defines a synchronisation between the controller and the environment on action α. The side condition makes sure that the action transition with the highest priority is taken. Note that an edge without an action label (i.e. an internal action) is also restricted by the environment in the sense that internal actions have lower priority than external actions.

The second rule defines a delay transition. The side condition makes sure that the delay is only possible as long as the environment cannot synchronise. This restriction implies maximal progress, the controller must react to the environment whenever possible and can only delay when the environment is silent.

Clearly the behaviour of the controller according to the refined semantics is deterministic. Safety properties in the design model are preserved since the behaviour defined by the deterministic semantics is included in the behaviour of the operational semantics. That is, all sequences of transitions (i.e. executions) of a design model for a controller according to the deterministic semantics can also be taken according to the operational semantics.

Note that according to the semantics, we may have automata that exhibit zeno-behaviours, that is, infinite sequences of action transitions within a finite time delay. Such automata correspond to non-implementable designs models, and should be discovered by schedulability analysis as non-schedulable. To simplify analysis, we adopt the syntactical restriction that all cycles in an automaton should contain at least one edge labelled with an action associated with a task.

Example 3.1 *We illustrate the difference between the deterministic and operational semantics by considering the automaton shown in Figure 3.2. For simplicity the example does not include an environment or any tasks. We have assigned priorities* ***H*** *(high) and* ***L*** *(low) to edges where needed.*

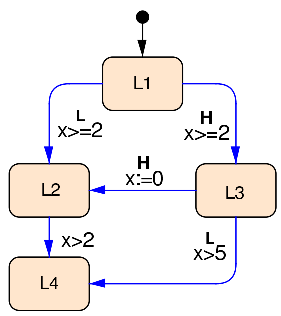


Figure 3.2: Timed Automata with priorities assigned to edges.

According to the operational semantics the automaton may exhibit three principal *behaviours which are characterised by the locations they pass and differ only in the delays. We exemplify them with these concrete traces:*

- $(L1, [x = 0]) \xrightarrow{2} (L1, [x = 2]) \xrightarrow{\tau} (L2, [x = 2]) \xrightarrow{\tau} (L4, [x = 2])$
- $(L1, [x = 0]) \xrightarrow{2} (L1, [x = 2]) \xrightarrow{\tau} (L3, [x = 2]) \xrightarrow{\tau} (L2, [x = 0]) \xrightarrow{2} (L2, [x = 2]) \xrightarrow{\tau} (L4, [x = 2])$
- $(L1, [x = 0]) \xrightarrow{2} (L1, [x = 2]) \xrightarrow{\tau} (L3, [x = 2]) \xrightarrow{3} (L2, [x = 5]) \xrightarrow{\tau} (L4, [x = 5])$

The deterministic semantics only leaves the second execution as possible. In the initial location the maximal progress assumption allow for only 2 time units delay, after which the transitions to locations L2 and L3 are enabled. Since the transition to L3 has higher priority it is taken and in location L3 the transition to L2 is already enabled, and must be taken immediately. In location L2 the automaton has to delay for 2 time units until the transition to L4 becomes enabled, and is taken immediately.

3.2 Synthesising executable code

The code generation transforms a design model to an executable program that will behave according to the deterministic semantics. We assume a generic target platform which guarantees the synchronous hypothesis and on which the associated tasks consumes their given computing time to execute. As the transformation preserves the deterministic semantics, the schedulability and other safety properties are preserved in the generated code when it is executed on the target platform.

We assume that the target platform runs an operating system that provides the following generic features:

- threads with unique priory levels,
- a scheduler based on fixed priority assignment,
- interrupt handling routines.

The first two requirements are needed to map the notion of tasks from the design model to he generated code. Tasks have unique priorities in the model, thus when they are mapped to threads in the implementation the threads must also have unique priorities. The last requirement is needed handle external events and to encode control automata, which is done by interrupts using interrupt handling routines.

The intended target platform is equipped with a single processor, but for the presentation we assume an execution model with two processors as shown in Figure 3.3. The code of the controller execute on a dedicated *control processor* and the task code execute on the *task processor*. The interaction between the processors is limited to the task queue and the shared variables. The control processor receives events from the environment and inserts the tasks into the task queue. The task processor executes the task at the head of the queue and updates the shared variables when a task is done.

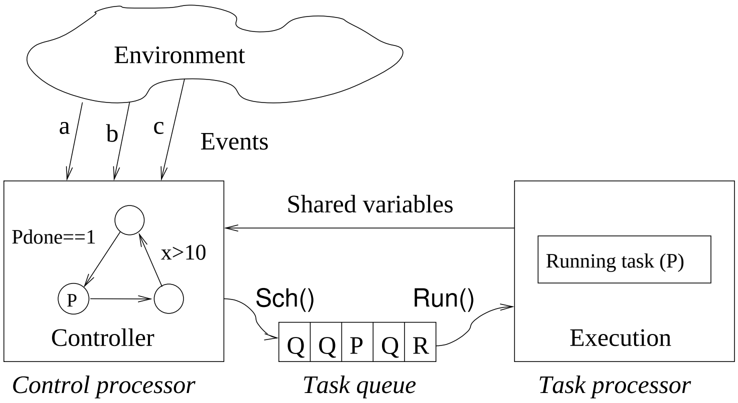


Figure 3.3: A logical view of an executing system, with a *control processor* handling events from the environment and a *task processor* executing task code. Communication between the processors is limited to the task queue and the shared variables.

Note that tasks may also interact with the environment but we do not illustrate this in the figure since, from the controllers view, the interface with the tasks are solely through the shared variables.

To realise the execution model on a single processor, we implement the code of the control automata as a separate thread with higher priority than all other threads.

3.2.1 Handling Tasks and Variables

Tasks are executed in threads, one thread for each task type. Scheduling and queue management is handled by the underlying operating system. We assume a bounded queue length, meaning that the memory allocated for the task queue can be fixed at compile-time and that no exception handling for queue overflow is needed at run time.

Data variables in the design model are mapped to global integer variables in the generated code. The code also reserve storage space in main memory for the contexts of released tasks. The analysis provide a bound on the space needed to store contexts so the memory can be allocated statically at compile-time.

Continuing the discussion in Section 2.1.1 we note that if task contexts are copied at release time more memory is needed since each instance of the task in the queue needs its own context. If the context on the other hand is copied when the task starts only one context for each task type is needed since only one instance of each task is running at a given time.

To encode clocks we use the difference between a global system clock sc and the last reset time. For each clock x in the design model let x_{reset} be an integer variable holding the system time of the last reset of that clock. The value of the clock is then $(sc - x_{reset})$, and a reset can be performed as $x_{reset} := sc$.

3.2.2 Encoding and Executing the Controller Automata

The Controller (c.f. Figure 3.3) is essentially an encoding of the structure of the control automata. We use four look-up tables and two functions, as shown in Figure 3.4, for the encoding. The static structure of the automata is encoded in the three look-up tables **edges**, **locations** and **actions**. The fourth table ACTIVE is dynamic and holds the set of currently active edges, i.e. edges leaving locations in the current location vector. Formally, when the controller is in state $(\bar{l}, \sigma, q)$ the table contains all edges $\left\{l_i \xrightarrow{g,\alpha,r} l_i' \in E\right\}$ such that l_i is an element in $\bar{l}$.

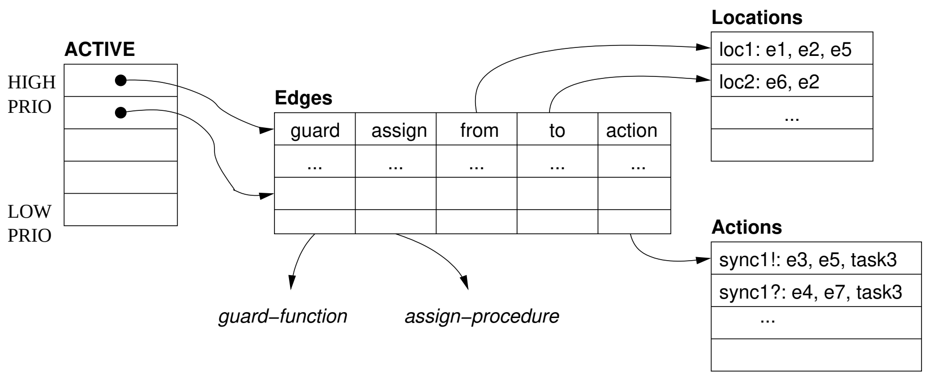


Figure 3.4: Data structures encoding the controller automata.

The table **edges** is sorted in priority order. For each edge there are five fields: *guard, assign, from, to* and *sync.* The *guard* and *assign* fields are references to functions that evaluate the guard and perform the assignments of the edge. The fields *from* and *to* are references into the table **locations**. The field *sync* is either empty if the edge has no action label or a reference into the table **actions**.

The table **locations** store, for each location, the edges leaving the location. The table **actions** stores, for each action label, references to all edges where the label occurs, and also any task associated with the label. Note that some of the information stored in the tables (such as the edges leaving a certain location) is intentionally duplicated to improve efficiency.

The list of active edges, ACTIVE, is managed by the code shown in Procedure 1. We use $sort$ to denote a sorting function that takes as input two parameters: a list of edges, and a function assigning unique priorities to edges, and returns as output a list of edges sorted according to the assigned priorities.

The pseudo-code in Procedure 1 is executed by the controller thread whenever an event (such as timeout or arrival of an external event) has occurred. When executing, no new events are processed and the timers are not updated, which means that the whole procedure is executed in a critical region. It also means that tasks are not able to update shared variables while the controller is executing.

Initially the list of active edges consists of all edges leading out from locations in the initial location vector. Then the procedure scans ACTIVE in priority order and evaluates the corresponding guards. If it finds a satisfied guard there

Procedure 1 Pseudo-code that executes the encoded controller.

Initially:
$\mathsf{ACTIVE} := \{l_i \xrightarrow{g,\alpha,u} l'_i | l_i = l_i^0\}$
$\mathsf{ACTIVE} := sort(\mathsf{ACTIVE}, \mathsf{Pr})$

Procedure:
```
START:
```
for each $l_i \xrightarrow{g,\alpha,u} l'_i$ in ACTIVE **do**
    if σ satisfies g **then**
        if $\alpha = \tau$ **then**
            $\sigma := u[\sigma]$
            remove $\{n \xrightarrow{g',\alpha',u'} n' | n = l_i\}$ from ACTIVE
            add $\{n \xrightarrow{g',\alpha',u'} n' | n = l'_i\}$ to ACTIVE
            $\mathsf{ACTIVE} := sort(\mathsf{ACTIVE}, \mathsf{Pr})$
            $L_{M(\alpha)} := V(M(\alpha))$
            $q := \mathsf{Sch}(M(\alpha) :: q)$
            goto `START`
        else
            if exists $l_j \xrightarrow{g',\bar{\alpha},u'} l'_j$ in ACTIVE such that σ satisfies g' and $i \neq j$ **then**
                if α is sending **then**
                    $\sigma := u[\sigma]$; $\sigma := u'[\sigma]$
                else
                    $\sigma := u'[\sigma]$; $\sigma := u[\sigma]$
                remove $\{n \xrightarrow{g',\alpha',u'} n' | n = l_i \vee n = l_j\}$ from ACTIVE
                add $\{n \xrightarrow{g',\alpha',u'} n' | n = l'_i \vee n = l'_j\}$ to ACTIVE
                $\mathsf{ACTIVE} := sort(\mathsf{ACTIVE}, \mathsf{Pr})$
                $L_{M(\alpha)} := V(M(\alpha))$; $L_{M(\bar{\alpha})} := V(M(\bar{\alpha}))$
                $q := \mathsf{Sch}(M(\alpha) :: M(\bar{\alpha}) :: q)$
                goto `START`
            fi
        fi
    fi
od

are two possibilities:

- no synchronisation – the assignment is performed and the information in the table **locations** is used to update the list of active edges and release any tasks associated with the edge.
- synchronisation – (i.e. the edge is labelled with an action label) the information in table **actions** is used to find an active edge belonging to another component in the network with complementary action label and satisfied guard. If such an edge is found, the compound transition is performed i.e. the assignments of the two edges are performed, ACTIVE is updated and the tasks associated with the action labels are released.

When a task is released the context is saved to global storage so that the task may use these values.

When a transition has been executed the procedure returns to `START` and re-examines the updated list of active edges to check for another transition to take. The procedure continues to execute action transitions as long as there exists enabled edges in ACTIVE. When there are no more enabled edges, the procedure terminates to allow the OS to schedule task threads. As an effect, the procedure implements a so-called *run-to-completion* step ensuring that the generated code has the maximal-progress behaviour assumed by the deterministic semantics (see Section 3.1). Note that since ACTIVE is always kept sorted and the procedure always scans from the head of the list, the implementation is also deterministic with respect to external actions.

Example 3.2 *We exemplify the encoding of a controller and its execution by the network shown in Figure 3.5. The network uses one integer variable i and synchronise on label* a*. The encoding is shown in Tables 3.1, 3.2 and 3.3. We assign priorities so that actions in Component A have priority over actions in Component B.*

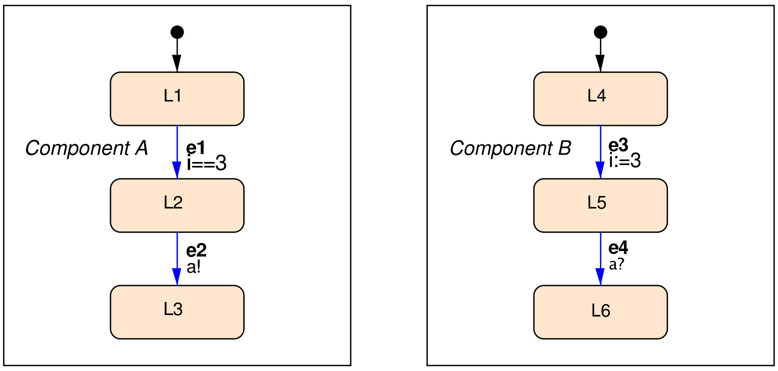


Figure 3.5: Two synchronising automata.

Initially

[1]Note that the action label is opposite to the label on the edge. We want to find the complementary edges, not the ones with the same label.

Table 3.1: Edges table.

name	guard	assign	from	to	action[1]
e1	$i == 3$	-	L1	L2	-1
e2	*true*	$i := 3$	L2	L3	a?
e3	*true*	-	L4	L5	-1
e4	*true*	-	L5	L6	a!

Initially the state of the network is $(\{L1, L4\}, [i = 0])$ *(where we ignore the task queue), which is represented by the (sorted) edges in* ACTIVE $= \{e1, e3\}$ *and a variable* $i = 0$.

First transition

We scan ACTIVE *in order and find that the guard of edge* $e1$ *is not satisfied in this state but the guard of edge* $e3$ *is* true*. So the assignment is performed and the information in Tables 3.1 and 3.2 is used to find that edge* $e3$ *should be replaced by* $e4$ *in* ACTIVE*. The new state is represented by the edges in* ACTIVE $= \{e1, e4\}$ *and the variable* $i = 3$.

Table 3.2: Location table.

location	outgoing
L1	e1
L2	e2
L3	-
L4	e3
L5	e4
L6	-

Table 3.3: Synchronisation table.

name	transitions
a!	e2
a?	e4

Second transition

We again scan ACTIVE *in order. This time* $e1$ *is enabled so it is taken. The new state, after replacement of* $e1$ *by* $e2$ *in* ACTIVE*, is represented by* ACTIVE $= \{e2, e4\}$ *and* $i = 3$.

Third transition

The third transition is a compound synchronisation transition. The first edge in ACTIVE, $e2$ *has a satisfied guard but is also labelled with the action label* $a?$*. In Table 3.3 we see that for* $a?$ *the edge* $e4$ *has the complementary label. The guard of* $e4$ *is satisfied so a compound transition is performed. Both* $e2$ *and* $e4$ *are removed from* ACTIVE, *thereby leaving it empty so that no more actions are possible.*

3.3 Correctness

In this section we argue for the correctness of the code generation by emphasising how the deterministic semantics of timed automata with tasks is realised in the generated code. The three components of a semantic state $(\bar{l}, \sigma, q)$ are mapped to the following parts of the generated code:

$\bar{l}$ — the source locations of the edges in ACTIVE (the list of active edges),

σ — integer variables for the data variables in σ, and the difference between reset-time and system time for the clocks in σ,

q — the scheduling queue of the operating system and global variables for contexts.

Note that there is no explicit representation of the elements in the location vector $\bar{l}$, instead the current configuration is implicitly represented by the edges in ACTIVE.

When the generated code starts to execute, the initial state $(\bar{l}_0, \sigma_0, q_0)$ is represented in the code as:

$\bar{l}_0$ — ACTIVE is initialised with the edges leading out from the initial locations,

σ_0 — data variables are initialised to their initial values and the reset-time of clocks are initialised to the system time at startup,

q_0 — the scheduling queue is empty which implies that no context is in use.

The transition rules defined by the deterministic semantics (c.f. Definition 3.1), are mapped in the following way:

Action — $(\bar{l}, \sigma, q) \xrightarrow{\alpha} (\bar{l}', \sigma', q')$ is handled by the code in Procedure 1. All edges that leads out from the source location are removed from ACTIVE and all edges that leads out from the target location are added to ACTIVE. Data variables are updated according to the assignments. Clocks are reset by assigning their reset-time to the current value of the system clock. The queue is increased when a task thread is started and the variables the task uses are copied to the tasks context.

Delay $(\bar{l}, \sigma, q) \xrightarrow{d} (\bar{l}, \sigma', q')$ correspond to execution of tasks. The location component of the state does not change during task execution. Data variables are updated by tasks when they finish. Clocks are updated by the delay d during task execution since the difference between the last reset-time and the system clock increases. The queue is reduced by removing the task at the head of the queue when it finish.

The refinements of the operational semantics introduced in Section 3.1 to resolve non-determinism are implemented as follows:

Priority Recall that priorities are assigned to components, action labels and edges so that all action transitions have unique priorities. The controller procedure evaluates the guards of the active edges in priority order from high to low. The first enabled action transition is taken. In the case of synchronisation the complementary enabled transition with the highest priority is taken. The order of the active edges is maintained after an action transition by resorting before proceeding.

Maximal progress The code has maximal-progress behaviour since, when a discrete transition has been taken the list of active edges is checked again. Only when no edge has a satisfied guard will the controller suspend and other threads with lower priorities (i.e. tasks) execute.

3.4 Optimising the generated code

The code generation described above can be improved by utilising the information gathered during analysis. We discuss a couple of optimisations of various complexity in this section.

Unreachable code With reachability analysis we can establish that a location in the design is unreachable. Under the assumption that the real environment behaves as the one used in analysis no code needs to be generated for the location or for edges leading there. If an action on any removed edge releases a task that is only released there we may even remove the code for that task.

With additional instrumentation we may also use reachability to discover unused edges. A location may have several incoming edges and all of them may not be used. To discover which edges are actually reachable we can annotate all edges with an assignment to a boolean variable, one variable for each edge. When a full state space exploration is performed all used edges will have their corresponding variable set to true. All edges whose variable is still false is unused and may be removed.

We can of course raise the question if unreachable locations or unused edges in the design are errors rather than sources for optimisation. One can argue that the location or edge was added to the design to fill a certain purpose. If it is unreachable this purpose may not be met and the fact that the location or edge is unreachable should indicate that something may be wrong.

Duplicated guards and assignments Another simple optimisation of code size is to collect all edges with the same guard and use only one guard index in the edges table for all these guards. For example, in Table 3.1 edges $e2$, $e3$ and $e4$ have all the guard *true*, which can be replaced by an index to one evaluation.

The same type of syntactical optimisation can be done for assignments.

Data domain When the full state-space search of the design model is performed to check for schedulability, we can at the same time gather information about the domains of the data variables. This information can be utilised when allocating memory for the data variables. I.e. if an integer variable only takes values in the range 1 to 4 there is no need to use a full 32 bit integer. How this optimisation performs will depend on both the hardware and the compiler. It may be inefficient to handle several small variables that does not follow byte or word boundaries when stored. The tradeof between speed and size depends on how good the compiler is and what support the hardware has.

3.5 Prototype for brickOS

The code generation described above has been implemented in a version of the Times-tool [AFM$^+$02] to generate code for the brickOS [Bri03] operating system – a small open source operating system for the Hitachi H8 processor – embedded into the LEGO® Mindstorms RCX control brick. The OS is implemented in C and for simplicity we use C as implementation language for tasks as well.

The Hitachi H8 processor in the RCX unit is equipped with 32 kB of RAM, which is a typical setup for the type of embedded systems that our code generation is intended for. The I/O interface consists of three sensor inputs, three actuator outputs, an infrared transceiver and a 5 character display.

In brickOS threads are separate control flows that are scheduled on the CPU by the operating system. The scheduler implements preemptive fixed priority scheduling and handles up to 20 priority levels. Threads with equal priorities are executed in a round robin fashion. This limits the number of task types in the design to at most 19, since each task type needs a unique priority level and the highest priority is reserved for the controller thread.

The OS provides so-called wake-up functions to program event handling. A wake-up function is a general mechanism provided by brickOS that lets a thread wait for a condition (such as the release of a semaphore, a timeout, a key-press etc.). To use a wake-up function the thread registers a boolean function that the scheduler executes when it looks for the next thread to run, which it does when the current thread is suspended or periodically every 20 ms.[2]

To make sure that the control procedure in Procedure 1 is executed at every event handling, we encode the entire control procedure as a wake-up function that always return false, and associate it to a thread at the highest priority level. In this way, the control procedure is ensured to be executed by the operating system (atomically and at OS priority level) just after every event handling, and just before the operating system determines the next task thread to be executed.

[2]The period of he scheduler (the time-slice) can be modified by recompiling brickOS.

For each task type a corresponding thread is created at boot time. The task threads contains a loop repeating a suspend call and the actual task body. The suspend call registers a wake-up function that checks for a non-zero value of an element in the integer array *release list*. This array contains one element for each task type which are incremented when an instance of the task type is released and decremented when an instance is finished. Note that since priorities are fixed, the release list is also a representation of the current state of the task queue.

Sensor readings in brickOS are available to the generated program as variables which are updated independently in the background by the operating system. Sensor variables are either read by the task code, or used directly in guards of the control automata. This means that checking if a transition is enabled becomes a condition only involving variables (external and other), i.e. we need no special treatment for external actions.

In the current prototype we associate tasks with locations instead of actions. The tasks are released when the location is entered. Such design models can easily be transformed into models with tasks associated to action labels by associating the task of a location with a new label on all edges leading to the location. Furthermore we impose the syntactic restriction that a control automaton may not use both a sending and a receiving action label with the same name (e.g. not both *sync*! and *sync*? in the same automaton). If this was allowed the automaton would be able to synchronise with itself!

3.6 Times tool

The tool Times [3] [AFM$^+$02] implements the analysis presented in Section 2.4 and the code generation presented in this chapter. The tool lets the user create and edit design models described as timed automata with tasks. In Figure 3.6 the edit view of Times is depicted. The right part of the window show the automaton just being edited. In the upper left part the tasks in the system are listed with their properties. In the lower left part the properties of the edited automaton and the variables are shown. The editor of Times also features a code editor for task code.

Times supports analysis by transforming the timed automata with tasks model into a model in the class of timed automata with bounded subtraction operations on clocks. The transformation also generates a scheduler automaton that is composed in parallel with the rest of the automata in the system, as described in Section 2.4. The scheduler automata is generated based on the scheduling policy of the system and the set of tasks. Its purposes are to ensure that the tasks are executed according to the scheduling policy, that the interface variables are updated when the tasks are finished and to indicate if a task fails to meet its deadline. A detailed description of the encoding can be found in [FPY02] and [FMPY03].

The execution of the system can be studied with the Simulator in Times. The Simulation view, shown in Figure 3.7, provides a Message Sequence Chart (upper right) to show the communication between the components in the system and a Gantt chart (lower right) to show execution of tasks. The left part of

[3] The tool is available at Times website: `http:\\www.timestool.com`.

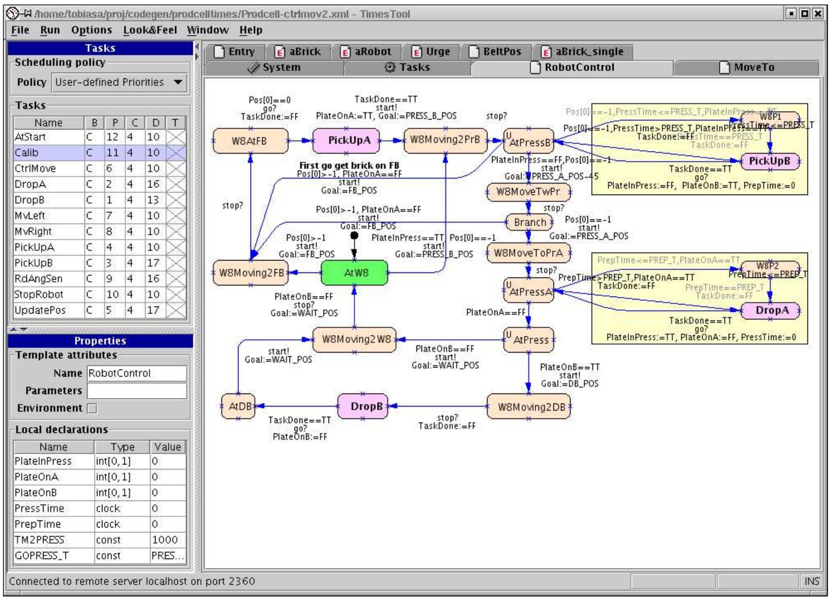


Figure 3.6: The edit view of Times tool.

the simulation view shows the enabled edges in the upper part and the current values of the data variables in the lower part.

As a final step in the design of a controller Times provides an implementation of the code generation described in this chapter. The generated C-code is targeted at the brickOS operating system running on a LEGO® Mindstorms RCX unit equipped with a Hitachi H8 processor. The generated code can be compiled by calling a compiler from within Times.

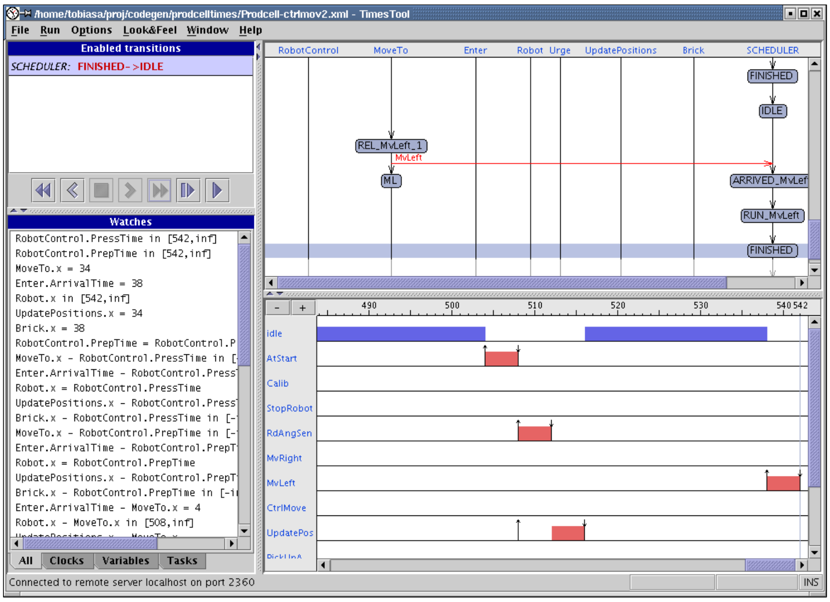


Figure 3.7: The Simulator view of Times.

Chapter 4

Animation of Hybrid Systems

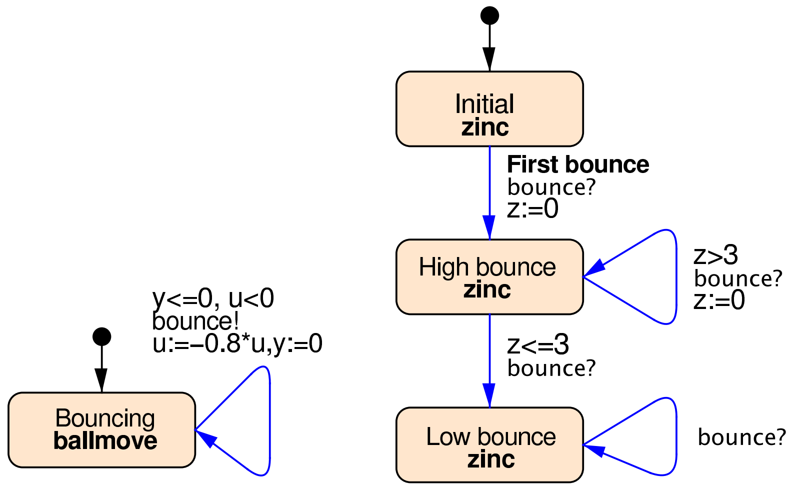
In this chapter we address the problem of exploring, before we put its implementation into use, if a design is appropriate for its intended task. We can use the analysis techniques presented in Section 2.4 to verify that the design fulfils a set of properties. Some of the properties (such as schedulability and boundedness) are necessary to perform code generation, but they do not say very much on the functionality of the system. We can devise other properties that may be more relevant for the functionality. But there is no easy way to determine if the set of properties we verify for the design covers all important requirements.

Instead we shall in this chapter develop a simulation environment, based on hybrid automata, for the TAT-designs. The idea is to make detailed models of the objects in the environment using hybrid automata and use the code generation to generate an executable simulation of the system. On top of this simulated system we place a visualisation layer that makes it possible to see what the system is doing and possibly to interact with it. With the help of the visualisation the designer can improve his or her understanding of the system.

4.1 Hybrid Systems

A hybrid system is a dynamical system that may contain both discrete and continuous components whose behaviour follows physical laws [Ant00]. We shall adopt hybrid automata [Hen96] as a basic model for such systems. A hybrid automata is a finite automata extended with real valued variables whose behaviour is determined by differential equations that are associated to the control locations. Timed automata can be seen as a special class of hybrid automata where the clocks are real valued variables with the differential equation $\dot{x} = 1$ associated to all locations for all clocks x.

The set of differential equations associated to locations is similar to the notion of tasks in TAT. This indicates the view we will take in this chapter, that the differential equations are special types of tasks with a restricted syntax. We will also discuss the semantics of hybrid automata in the context of timed automata with tasks. First we introduce the notation of hybrid automata by an example:



(a) Ball (b) Monitor

Figure 4.1: Hybrid automata modelling a bouncing ball and a monitor.

Example 4.1 *Let us consider a ball bouncing on a floor. While the ball is moving up and down it is best described by differential equations. But when it hits the floor and rebounds the direction of movement is suddenly inverted and a discrete transition is a better (abstract) description*[1]*.*

In Figure 4.1 the model of a bouncing ball and a monitor automaton is depicted. The system contains two real-valued variables y, z and an auxiliary variable u defined as the first time derivative of y, i.e. the speed of y. We interpret the y variable as the vertical height of the balls centre of mass.

Initially the variables have the values $y = 20$, $u = 0$, $z = 0$. I.e. the ball is at rest on height 20. The variables are updated by the equations (listed in Table 4.1) associated with the locations. The variable z changes with constant speed ($\dot{z} = 1$). In the vertical direction the ball accelerates downward ($\ddot{y} = \dot{u} = -9.8$). When $y = 0$ the ball hits the floor where it makes a discrete transition and bounce back up. In the bounce some of the energy is lost so the speed is reduced to 80% of the initial (u is assigned to $-0.8 \times u$). We assume that the floor is touch sensitive and that a signal is sent to the monitor, using the action bounce!*, every time the floor is touched.*

The monitor measures the time interval between bounces. Initially the monitor waits for the first bounce at which it proceeds to the location High bounce*. As long as the bounces takes more than 3 time units it stays there. But when a bounce took less than or equal to 3 time units the monitor proceeds to* Low bounce*. Note that the only variable in the monitor, z, always has constant speed, so the monitor can be interpreted as a timed automaton.*

[1]We could make a more detailed and purely continuous mathematical model of how the ball is compressed and how the energy is converted in the bounce, but that may be a to detailed description for our purpose.

Table 4.1: Equations in locations of automata in Figure 4.1.

Equations in **ballmove**	Equations in **zinc**
$\dot{y} = u$	$\dot{z} = 1$
$\dot{u} = -9.8$	

4.1.1 Syntax

The syntax of hybrid automata is similar to that of timed automata with tasks. The main difference is that clocks are replaced by real-valued variables and tasks with differential equations.

Let $\mathcal{A}ct$ be the set of action labels, and X be a set of real-valued variables ranged over by x, y, z etc. including a time variable t. We use $\dot{x}$ to denote the first time derivative (rate) of x and write $\dot{X}$ to denote the set of all first derivatives of variables in X. The variable t represents the global time with a rate that is invariably $\dot{t} = 1$. The values of $\dot{x}$ are determined by differential equations of the form $\dot{x} = f_x(X \cup \dot{X})$ where f_x is a function over $X \cup \dot{X}$. Let $\mathcal{F}$, ranged over by f, denote the set of such functions.

We assume a set of predicates over the values in X. We use $\mathcal{G}$, ranged over by g, h etc., to denote the set of boolean combinations of the predicates, called guards.

At discrete transitions we use assignments to manipulate variables. The assignments are of the form: $x' := \gamma(X \cup \dot{X})$, where x' denotes the value after the discrete transition. Assignments take the current values of of the variables as parameters and update all variables simultaneously. We use Γ, ranged over by γ, to stand for the set of assignments.

Definition 4.1 *A Hybrid Automata over real variables X, differential equations $\mathcal{F}$, guards $\mathcal{G}$, actions $\mathcal{A}ct$ and assignments Γ is a tuple $\langle N, E, I, M, l_0, X_0 \rangle$ where:*

- *N is a finite set of locations ranged over by l, m, n,*
- *$l_0 \in N$ is the initial location,*
- *$E \subseteq N \times \mathcal{G} \times \mathcal{A}ct \times \Gamma \times N$ is the set of edges,*
- *$I : N \to \mathcal{G}$ is a function assigning each location with an invariant,*
- *$M : N \to 2^{\mathcal{F}}$ is a function assigning locations with differential equations (tasks),*
- *$l_0 \in N$ is the initial location, and*
- *X_0 is the initial variable assignment.*

In the same manner as for timed automata with tasks, see Section 2.3, we extend hybrid automata to networks of hybrid automata.

4.1.2 Semantics

The semantics of networks of hybrid automata is similar to that of networks of timed automata with tasks. We view the differential equations associated with the locations as specialised task types. As for timed automata hybrid automata is semantically a labelled transition system with two types of transitions, actions and delays. On action transitions tasks are released for execution and on delay transitions tasks are executed.

The major difference from timed automata with tasks is that we don't consider scheduling of the differential equation tasks. The tasks are only executed as long as the automaton stays in the corresponding location. Therefor there is no need for a queue and a semantic state of a network of hybrid automata is a pair $(\bar{l}, \sigma)$. Where $\bar{l}$ again is a location vector and σ is a variable valuation.

A *variable valuation* for hybrid automata is a mapping from variables X to the reals. For a variable valuation σ and a delay Δ (a positive real) , $\sigma + \Delta$ denotes the variable valuation such that

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

That is, the effect on the variables as determined by the differential equations after the delay.

A *discrete assignment* is a transformation of a variable valuation according to the functions in an assignment. Let $Val(e, \sigma)$ denote the value of expression $(x := e) \in \Gamma$ evaluated in σ. Given a valuation σ and a discrete assignment γ we use $\gamma[\sigma]$ to denote the new variable valuation σ' where:

$$\begin{cases} \sigma'(x) = Val(e, \sigma) & \text{if } (x := e) \in \gamma \\ \sigma'(x) = \sigma(x) & \text{otherwise} \end{cases}$$

Given a guard $g \in \mathcal{G}$ and a variable assignment σ, $\sigma \models g$ denotes that the guard is satisfied in σ.

Definition 4.2 *(Hybrid automata semantics) The semantics of a network of hybrid automata $\langle N, E, I, M, \bar{l}_0, X_0 \rangle$ is a labelled transition system with the initial state $(\bar{l}_0, \sigma_0)$ where the transition relations are defined by the following three rules.*

- $(\bar{l}, \sigma) \xrightarrow{\alpha} (\bar{l}[l'_i/l_i], \gamma_i[\sigma])$ *if* $l_i \overset{g,\alpha,r}{\longrightarrow} l'_i$ *and* $\sigma \models g_i$
- $(\bar{l}, \sigma) \xrightarrow{\tau} (\bar{l}[l'_i/l_i, l'_j/l_j], \gamma_j[\gamma_i[\sigma]])$ *if* $l_i \overset{g_i,\alpha!,r_i}{\longrightarrow} l'_i$, $l_j \overset{g_j,\alpha?,r_j}{\longrightarrow} l'_j$, $\sigma \models g_i$, $\sigma \models g_j$ *and* $i \neq j$
- $(\bar{l}, \sigma) \xrightarrow{\Delta} (\bar{l}, \sigma + \Delta)$ *if* $I(\bar{l})(\sigma + \Delta)$

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

4.1.3 Discrete semantics

The operational semantics for hybrid automata in Definition 4.2 defines how a hybrid automaton behaves in every real-valued time point with arbitrarily fine precision.

There is no practical way to implement the full operational semantics. Instead a "sampling" technique is needed to analyse or execute the system. We examine the system at discrete time points chosen to approximate the full system behaviour.

Hence we adopt a time-step semantics called the δ-semantics relativised by the time granularity δ which describes how the hybrid system shall behave in every δ time units. In practical applications the time granularity δ is chosen according to the nature of designed system and the differential equations involved. The time granularity should be shorter than any time interval used by the designed system, in for example guards, and should be short for rapidly changing functions since a smaller granularity gives better precision.

The discrete tick semantics is simply the operational semantics defined above but with the limitation that every delay step has exactly length δ.

Definition 4.3 *(Discrete semantics for hybrid automata) Given a network of hybrid automata $\langle N, E, I, M, \bar{l}_0, X_0 \rangle$ and a time granularity δ the discrete semantics is a restriction of the operational semantics (c.f. Definition 4.2) with the following transition rules:*

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

where χ denote time steps.

We refine the discrete semantics for hybrid automata in the same manner as with timed automata with tasks. That is, we adopt maximal progress and define priorities for action transitions. As a result we get a *discrete deterministic semantics* for hybrid automata that we use as the basis for code generation.

4.2 Implementation

The discrete deterministic semantics for hybrid automata has been integrated with the code generation in Times. The code generated for hybrid automata has the same structure as code for embedded systems but with the limitation that a task can only describe differential equations.

At the heart of the implementation we have adopted an existing solver for initial value problems for systems of ordinary differential equations (ODE:s). The solver is called CVODE [CH96] and is freely available with source code and extensive documentation.

We use CVODE as a kind of execution unit (task processor) for the differential equation tasks. Each task is transformed to a C function that defines the right hand side of the differential equations and are supplied to the solver. At each time tick the solver provides a vector of values of the variables which are used by the controller code.

The mathematical formulation of an initial value ODE problem is:

$$\dot{\mathbf{x}} = f(t, \mathbf{x}), \ \mathbf{x}(t_0) = \mathbf{x}_0, \ \mathbf{x} \in \mathbb{R}^N \tag{4.1}$$

where we note that the derivatives are only of first order. Problems containing higher order differential equations can be transformed to a system of first order equations. The solver calculates numerical solutions to 4.1 as discrete values x_n at time points t_n.

CVODE provides several methods for solving ODEs, suitable for different kinds of problems. Since we aim at general usage of the simulator we cannot assume any properties of the system. Therefor we only apply the full dense solver to the problems and assume that the system is well behaved (non-stiff in numerical analysis terminology).

Execution of hybrid automata The execution of networks of hybrid automata is performed in a step-wise manner. Initially a solver is created for the task in the initial location of each component in the network.

In each step the solver for each component is called to calculate the values of the variables after one δ-tick. Then the controller (c.f. Procedure 1) is called to check for any action transitions. Recall that the controller performs a run-to-completion step, so several components may change locations. When the controller procedure returns new solvers are created for the tasks in components that have changed location (if any). Then a new step is performed with a call to CVODE followed by a call to the controller, and so on.

It is worth pointing out that the tick length δ is independent of the internal step size used by the ODE solver. The solver automatically chooses an appropriate internal step size according to the function calculated and parameters for acceptable local errors set by the user. From the solvers point of view the tick intervals can be seen as observation or sampling points.

At the boundary of guards The step-wise execution of the hybrid automata provides good approximations of the real valued variables during delay transitions. But as soon as a guard becomes satisfied there is a risk of “over-shoot”. The problem is that the guard may have been true somewhere between two ticks. If the execution continues from the new values a small error is introduced that could grow as the execution continues.

The problem is illustrated in Figure 4.2 where two consecutive values ($\mathbf{x}_n$ and $\mathbf{x}_{n+1}$) for the two dimensional variable $\mathbf{x} = (x_1, x_2)$ is plotted. In the first tick the guard (shown as a grey area) is not satisfied, but in the next it is. If the execution continues from $\mathbf{x}_{n+1}$ the trajectory is different than if it continues from the boundary of the guard.

To solve this problem we must find the point between two ticks where the guard just becomes true, that is the boundary of the guard. The solution that is implemented in the simulator is to successively halve the interval between $\mathbf{x}_n$ and $\mathbf{x}_{n+1}$ until the difference is smaller than a choosen tolerance and finally take the value just inside the boundary of the first satisfied guard.

Given the variable values y_i and y_{i-1} in two consecutive ticks and a tolerance the interval halving algorithm goes like this:

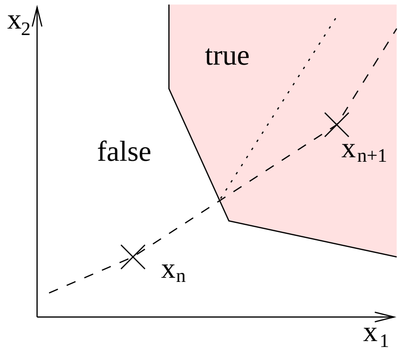


Figure 4.2: Two consecutive values and a guard.

---

repeat
  $y_{tmp} \leftarrow (y_i - y_{i-1})/2$
  if any_guard(y_{tmp}) **then**
    $y_i \leftarrow y_{tmp}$
  else
    $y_{i-1} \leftarrow y_{tmp}$
  end if
until $|y_i - y_{i-1}|$ < tolerance

---

where *any_guard* is a function that checks if any guard is true at a given point, y_{tmp} is a temporary variable and $|.|$ denote the distance between two points. The algorithm assume an approximately linear behaviour between the two points since the new value is found on the line connecting them.

4.3 Examples

In this section we illustrate the usage of the implemented hybrid automata animator by a couple of examples. First we discuss the bouncing ball introduced in Example 4.1. We also show a simple example where a billiard ball is rolling on the surface of a table.

4.3.1 Bouncing Ball

Recall the hybrid automata and the monitor in Figure 4.1. The ball is dropped from a height of 20 m with a small forward speed of 1 m/s. The ball accelerates downward until it hits the ground where it rebounds and move upward until the gravitational force has retarded the motion and it starts to fall again. In every bounce it looses some energy (simplified to reducing the vertical speed to 80%).

From the hybrid automata code has been generated that, when executed, outputs the values of the variables at the tick intervals. This output is drawn in Figure 4.3. In the graph two variables are drawn; *y* of BALL and *z* of MONITOR. The variable *y* indicates the height of the ball and *z* is used by the MONITOR to

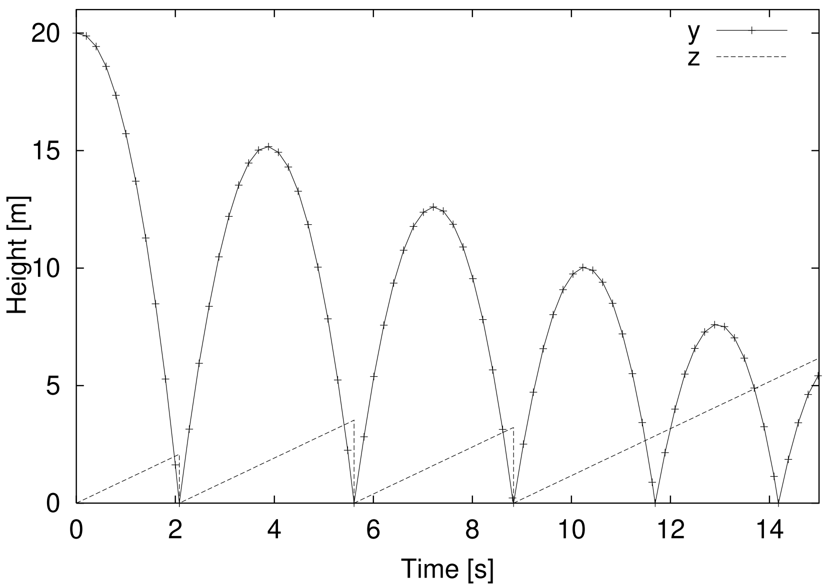


Figure 4.3: Bouncing ball.

measure the time between bounces. In the graph we see that the ball hits the floor for the first time at time slightly more than 2. At this time the monitor goes from Initial to High bounce. The monitor stays in High bounce for the next three bounces which takes more than 3 seconds to complete. But after the third which take less the MONITOR proceeds to Low bounce. On this transition z is not reset, so from then on z grows unbounded.

4.3.2 Billiard Ball

As a further example we consider a billiard ball rolling on a square table surrounded by low walls and with one hole in the surface. We consider this problem in a idealised version without friction and with perfect collisions where no energy is lost, only the direction changes.

The table is 4 by 4 m with a 1 by 0.5 m hole at one side. We assign one corner of the table as the origin of an orthogonal coordinate system with axis x and y. The ball is initially at position $x = 0$ and $y = 1$. The speed of the ball is in x direction $\dot{x} = c$ and in y direction $\dot{y} = d$ where c and d are constants. The ball rolls on the table and when it hits the wall the speed in one direction is inverted. If the ball falls into the hole it stops.

The movement of the ball can be described by the hybrid automata in Figure 4.4. There are two locations, one representing that the ball is rolling and one that it has fallen into the hole. Associated to the location Rolling there is a task, roll, which updates the position according to the current values of the constants c and d. The equations in roll is shown in Table 4.2. The location Goal does not have any associated task since the movement of the ball has stopped, so no variables need to be updated. The hole is represented by the guard of the

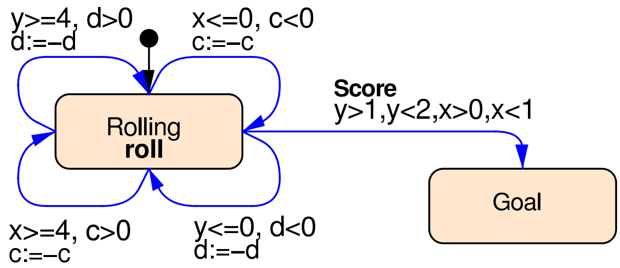


Figure 4.4: Billiard ball motion.

Table 4.2: Equations in location of automaton in Figure 4.4.

Equations in **roll**
$\dot{x} = c$
$\dot{y} = d$

edge **Score**.

Assuming that x and y are initially in the intervals $0 \leq x \leq 4$ and $0 \leq y \leq 4$, the trajectory of the ball on the table is determined by the automaton. In Figure 4.5 the trajectory of the ball with initial position $(0, 1)$ and constants $c = 1.5$ and $d = 1$ is shown. The grey area represent the hole. Since the ball indeed hits the hole when it follows this trajectory it stops just inside the boundary of the guard.

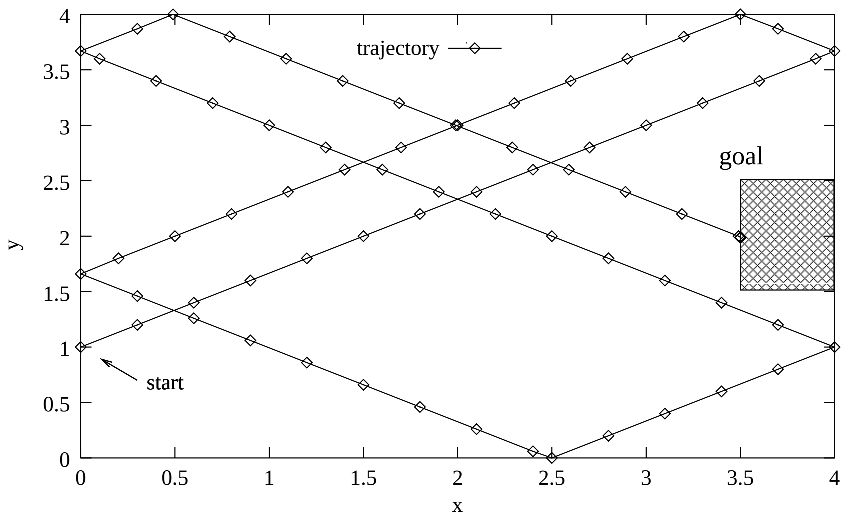


Figure 4.5: Trajectory of ball rolling on table without friction and with constant speed.

Chapter 5

Modelling and design of a production cell

In this chapter, we show how to use the design language presented in Chapter 2 to model and design the control software for a production cell. The production cell is a model of an industrial unit in a metal plate processing plant in Karlsruhe. A model of the production cell was developed by FZI in Karlsruhe [LL95] as a benchmark example of a concurrent and safety-critical system with real-time properties. In [LL95] several approaches to develop system software for the production cell based on different formal methods where presented.

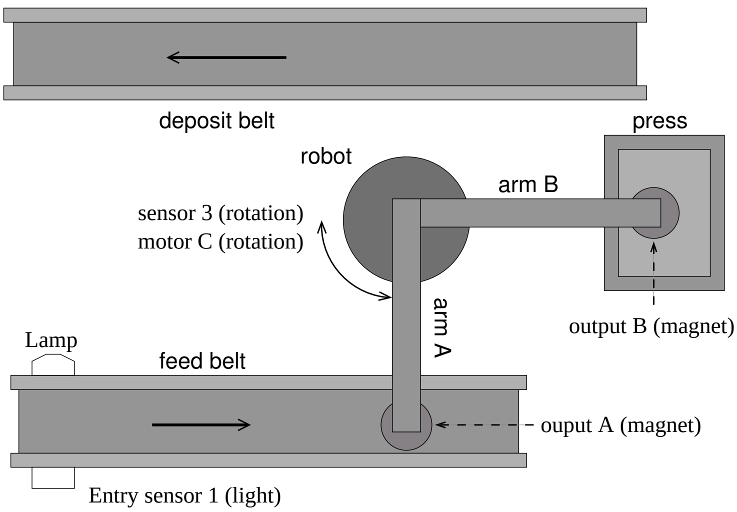


Figure 5.1: The LEGO model of the Production Cell.

In this chapter we use a LEGO® model of the production cell based on the model developed by FZI but with some simplifications. The LEGO® model of the production cell consists of four subsystems, a *feed belt*, a *robot*, a *press* and a *deposit belt* as shown in Figure 5.1. It does not included the elevating rotary table after the feed belt or the crane between the deposit belt and the feed belt of the original model. The metal plates in the original production cell are, in our case, represented by LEGO-bricks with metal plates glued to the top. The metal plates enables the robot, that have electro-magnets mounted at the end of its two arms, to loft and hold the bricks. The bricks enter the system at the beginning of the feed belt, then they are handled by the subsystems as follows:

The *feed belt* transports the bricks from the entry position to the end of the belt where the first robot arm (*arm A*) may pick them up. The robot then rotates so that arm A is directed at the *press,* where the brick can be dropped.

In the *press* the brick is forged for some time. In the original industrial setting the forging is the main activity of the cell. We are more interested in the movement of the bricks within the system, so we ignore the forging and leave the brick in the press for a time corresponding to its forging.

When the brick has been forged the robot can pick it up by positioning *arm B* over the press. With the brick attached to arm B the robot rotates so that arm B is in position to drop the brick onto the *deposit belt*. Finally the deposit belt transports the brick out of the system.

Our production cell is controlled by a LEGO® Mindstorms unit, a device that can be connected to sensors and motors to control a LEGO® model. Two of the three sensor ports are used, one to connect the light sensor positioned at the beginning of the feed belt and one to connect the rotation sensor at the robots rotation axis. Opposing the light sensor there is a small lamp mounted. When a brick passes between the lamp and the sensor the signal level from the light sensor drops and then rises again when the brick has passed. The three actuator ports are all connected to the robot, one to control the rotation and one each for the magnets on the arms. The press and the deposit belt do not have any sensors or actuators in the LEGO® model.

5.1 Designing the Controller

The main objective of the controller is to schedule the robot so that the bricks are moved from the feed belt to the press and on to the deposit belt. This must be achieved while fulfilling the real-time requirements imposed by the non-stopping feed belt. When a brick enters the system on the belt the robot must pick it up at the end of the belt or the brick falls off.

The controller for the production cell consists of two sub-controllers, one controls the feed belt and the other controls the robot subsystem. Each sub-controller consists of two cooperating automata.

The structure of the controllers are shown in Figure 5.2. The grey box to the left shows the three actuator ports that are connected to the physical environment. The grey box at the bottom shows the two sensor ports. The rectangular boxes represent four named control automata.

The automata interact with the environment via tasks, which are shown in the figure as labels in italic font. The interaction between the automata is handled with shared variables, channels and tasks. The *Robot Controller* uses the shared variables *Goal* and *Pos*, and two actions *start* and *stop*. The *Feed Belt Controller* uses the variable *Pos* and the tasks *AtStart* and *UpdatePos.*

The sub-controllers communicate via the shared integer array *Pos*. This

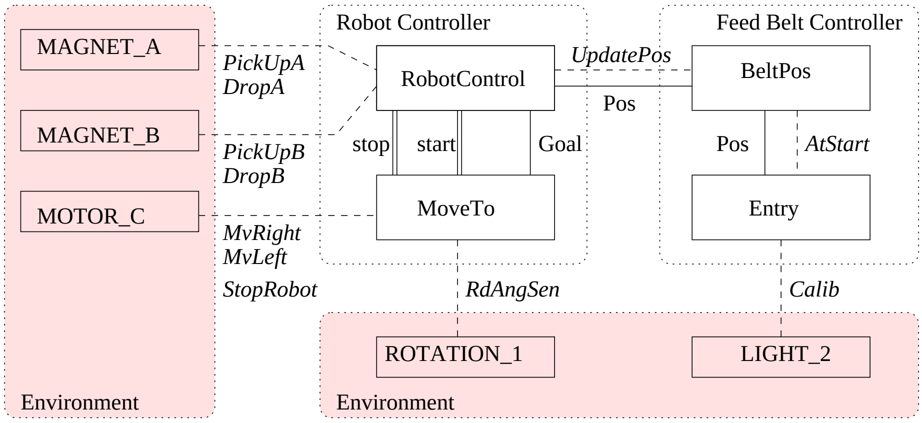


Figure 5.2: Structure of controller. Tasks are indicated as labels in italic font on dashed lines. Single line indicate shared variable, double line synchronisation labels.

array is used by the feed belt controller to keep track of the position of the bricks on the belt. If $Pos[0] > -1$ it means that there is a brick on the feed belt, if $Pos[0] = -1$ there is no brick on the belt. And if $Pos[0] = 0$ there is a brick at the position where it may be picked up by the magnet on *arm A*.

Figure 5.3 show a schematic picture of how the positions on the belt correspond to distance values in the array. There are two bricks on the belt at distances 2 and 6 from the pick up position. The distances are stored in the array and the content corresponding to the figure is $Pos = [2, 6, -1, -1, ...]$.

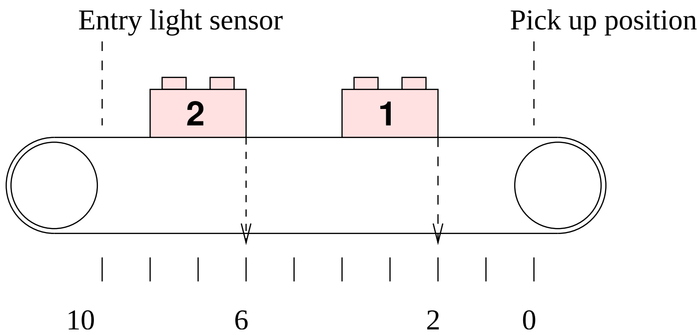


Figure 5.3: Feed belt with two bricks and the entry and pick-up positions marked.

5.1.1 Robot Controller model

The robot controller consists of two automata, ROBOTCONTROL and MOVETO. The overall planning and decisions takes place in ROBOTCONTROL while MOVETO act as a sub-procedure that executes a rotation of the robot to a specified position.

Whenever ROBOTCONTROL decides that the robot should move it sets the shared variable *Goal* and synchronises with MOVETO on the action start. When MOVETO has executed the rotation it synchronises with ROBOTCONTROL again on the action stop. While MOVETO performs the rotation ROBOTCONTROL is inactive without changing its location.

ROBOTCONTROL communicates with the feed belt controller via the first element of *Pos*.

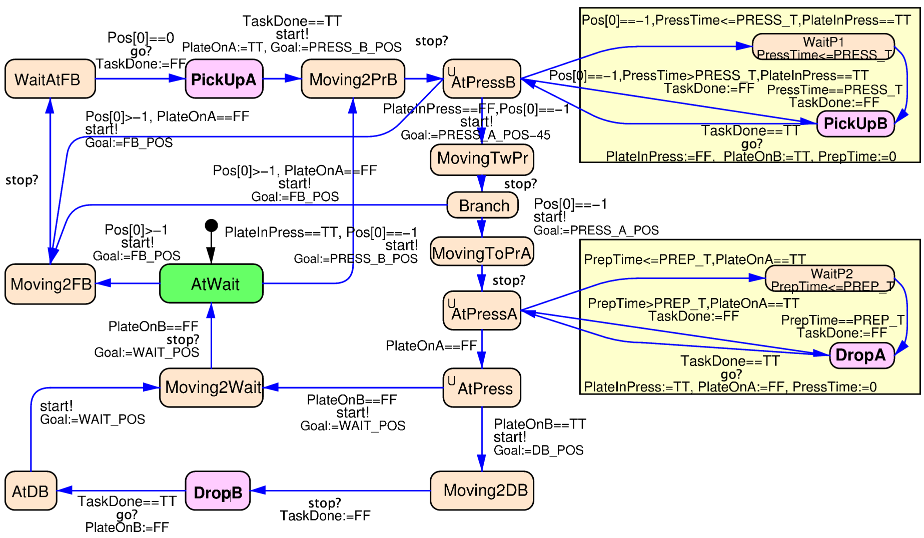


Figure 5.4: The automaton ROBOTCONTROL of the robot controller.

Automaton RobotControl The automaton ROBOTCONTROL, which is shown in Figure 5.4, uses three boolean variables to keep track of the state of the brick processing. *PlateInPress* indicates that there is a brick in the press and *PlateOnA* and *PlateOnB* indicates that a brick is on arm A and B respectively. Note that there is no sensor feedback to indicate if a pickup succeeded. The shared boolean variable *TaskDone* is used to signal that a released task has finished execution.

The automaton also uses two clocks; *PressTime* to measure the time the press needs to forge a plate and *PrepareTime* to measure the time it needs to recover afterwards. The constants `PREPARE_T` and `PRESS_T` hold these times. The Goal positions communicated to MOVETO are held in the constants `WAIT_POS`, `FB_POS`, `PRESS_A_POS`, `PRESS_B_POS` and `DB_POS`.

The normal operation of the production cell takes ROBOTCONTROL through a tour in the automaton starting in the initial location AtWait. When a brick has arrived on the feed belt ROBOTCONTROL synchronises with MOVETO and proceeds to location Moving2FB where it stays while the robot rotates to the feed belt. The automaton then continues through WaitAtFB to PickUpA where the task PickUpA is released.

After the pickup ROBOTCONTROL proceeds through Moving2PrB to AtPressB where the state variable *PlateInPress* is used to decide if a detour to

PickUpB is needed to lift a brick with *arm B*. If it is not needed the automaton continues through MovingTwPr,Branch and MovingToPrA to AtPressA where a new detour can be taken to DropA where the brick is dropped on the press.

While the press forges the plate the robot moves back to the waiting position and ROBOTCONTROL returns to the initial location. If no new brick has entered the system the controller goes to AtPressB via MovingToPrB where it waits for the brick in the press to be ready to be picked up by *arm B*. With the brick attached to the arm the controller moves down to MovingToDB where the robot rotates toward the deposit belt where the brick is dropped (in location DropB by task DropB).

Automaton MoveTo The automaton MOVETO, that controls the rotation of the robot, is shown in Figure 5.5. When the robot does not move the automaton stays in the initial location Entry. As soon as ROBOTCONTROL synchronises on start MOVETO proceeds to Read where the task RdAngSen is released to read the value of the angle sensor and store it in the variable *RobotAngle*. Depending on the measured value the robot should move right, left or not at all. This is decided by the guards on the edges to MR, ML and Entry respectively. If the robot must rotate to reach its goal the motor is started by tasks MvRight or MvLeft. When the goal is reached, MOVETO goes to SR where the task StopRobot is released to stop the motor.

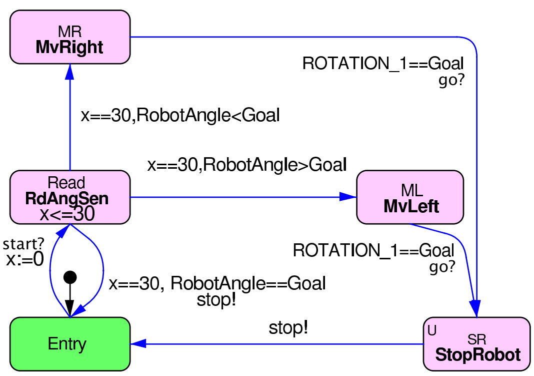


Figure 5.5: The MOVETO process that governs the movement of the robot.

5.1.2 The Feed Belt model

The design of the feed belt controller consists of two automata. The automaton ENTRY, shown in Figure 5.6, handles the light sensor at the start of the belt to detect arrival of bricks to the system. The automaton BELTPOSITION handles the internal representation of the movement of the bricks on the belt.

Automaton Entry The automaton ENTRY is initially in location Calib which has the task Calib associated. This task performs calibration of the background light by calculating the mean value of the light sensor reading during 500 ms.

Table 5.1: Constants used in controller and environment models

Constant	value	type	used in
WAIT_POS	45	angle	ROBOTCONTROL
FB_POS	0	angle	ROBOTCONTROL
PRESS_A_POS	90	angle	ROBOTCONTROL
PRESS_B_POS	0	angle	ROBOTCONTROL
DB_POS	90	angle	ROBOTCONTROL
PREPARE_T	200	time	ROBOTCONTROL
PRESS_T	250	time	ROBOTCONTROL
SAFE_CALIB	4000	time	ENTRY
BELT_LENGTH	10	length	AtStart
SENSORLOW	0	light	BRICK
SENSORHIGH	100	light	BRICK
ATSENSOR_T	500	time	BRICK
BETWEEN_T	3000	time	BRICK
ATPICKUP_T	500	time	BRICK
WAIT_T	2000	time	BRICK

When finished the task stores the calculated value in the variable *normalLight* and sets the boolean variable *EntryTaskDone* to true.

The automaton proceeds to WaitFrontEdge where it waits until the light sensor goes below 80% of the calibrated background light. When that happens it is interpreted as the first edge of the brick and the automaton continues to WaitBackEdge where it waits for the whole brick to pass through. When the light sensor again rises above 80% of the background light the automaton takes a transition to AtStart where the task AtStart, which inserts an element into the array *Pos*, is released.

The clock *ArrivalTime* measures the time since a brick last entered the system. A new brick should not be allowed to enter too close after the previous one or the robot will fail to handle both of them. During the time when bricks are not accepted a beep sound[1] is generated (represented by the assignment to the variable BEEP). The delay in AtStart enforced by the guard $\mathrm{ArrivalTime} >=$ SAFE_CALIB ensures that the total waiting time in AtStart and Calib sums up to the safety interval. The constant SAFE_CALIB is obtained by experiments.

Automaton BeltPosition The automaton BELTPOSITION, shown in Figure 5.7, handles the internal representation of the movement of the bricks on the belt. It uses the array *Pos* to keep track of the positions of the bricks. Initially, and when there are no bricks on the feed belt, the automaton stays in location Idle. As soon as there is a brick on the belt (indicated by Pos[0]>-1) it moves to location Active and releases the task UpdatePos. In Active, the task UpdatePos is released periodically to update the elements in *Pos*. When there is no more

[1]The beep is a signal to a human operator. In another setting, the automaton could send a signal to for example a gate that stop bricks from arriving.

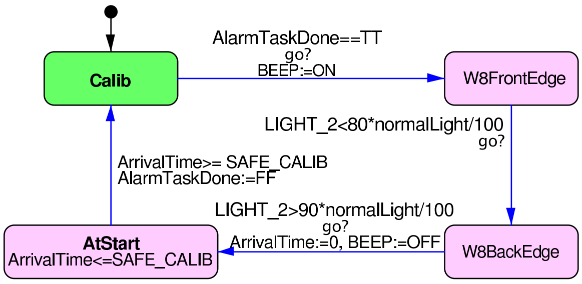


Figure 5.6: The automaton ENTRY handles the entry sensor in the feed belt controller.

bricks on the belt the automaton returns to Idle.

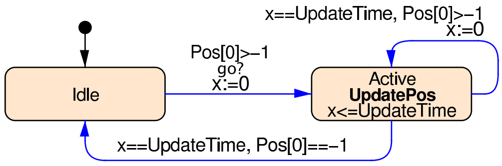


Figure 5.7: The BELTPOSITION process of the feed belt controller.

5.1.3 Tasks in the production cell

The tasks released by the automata in the production cell controller are summarised in the table 5.2 below. The parameters needed by the analysis are tabulated in columns C (computation time), D (deadline) and P (priority). The assignments performed by the tasks and a short description of the task is given in the column *Description and assignments*. Note that only the task interface is shown in the table, the full code is listed in Appendix A.

Table 5.2: Summary of tasks in production cell controller

Name	C	D	P	Description and assignments
		ENTRY		

AtStart	4	10	12	Append values to Pos:
				`Pos[numBricks]:=BELT_LENGTH,`
				`numBricks:=numBricks+1`

Calib	4	10	11	Read a calibrated value for background light:
				`EntryTaskDone:=TT`
				`normalLight:=100`

		BELTPOSITION		
UpdatePos	5	17	5	Update values in Pos: $\forall$`i Pos[i]:=(Pos[i]`$\geq$`0?Pos[i]-1:-1)`

Table 5.2: Summary of tasks in production cell controller

Name	C	D	P	Description and assignments
		ROBOTCONTROL		
DropA	4	16	2	De-activate magnet on arm A: `TaskDone:=TT`
DropB	4	13	1	De-activate magnet on arm B: `TaskDone:=TT`

PickUpA	4	10	4	Activate magnet A. Remove value from Pos:
				`Pos[i]:=Pos[i+1]`
				`Pos[BRICKS-1]:=-1`
				`numBricks:=numBricks-1`
				`TaskDone:=TT`

PickUpB	4	17	3	Activate magnet on arm B: `TaskDone:=TT`
		MOVETO		
MvRight	4	10	8	Start motor to rotate robot rightwards: `ROB_DIR:=CCW`
MvLeft	4	10	7	Start motor to rotate robot leftwards: `ROB_DIR:=CW`
RdAngSen	4	16	9	Read rotation sensor and convert to degrees: `RobotAngle:=ROTATION_1`
StopRobot	4	10	10	Stop rotation motor: `ROB_DIR:=STOP`

5.2 Analysis

The behaviour of the production cell has been analysed with Times in order to establish schedulability and other properties. In the analysis we have used timed automata models of the objects in the environment i.e. the physical feed belt and robot. Times transforms the design models into timed automata with subtractions in the clock variables and construct a scheduler automata. The analysis of the network of automata consisting of the environment models, the scheduler and the transformed design models are performed using reachability analysis as described in Section 2.4.

Properties are described using a limited subset of timed CTL using the following grammar:

$$
\begin{aligned}
query &\rightarrow {'}\forall\Box{'} prop\_expr \,|\, {'}\forall\Diamond{'} prop\_expr \,|\, {'}\exists\Box{'} prop\_expr \,|\, {'}\exists\Diamond{'} prop\_expr \,| \\
& \quad\; prop\_expr\, {'}-\,>{'}\, prop\_expr \\
prop\_expr &\rightarrow property \,|\, {'}not{'}\, prop\_expr \,|\, {'}({'}\, property\, {'}){'} \,| \\
& \quad\; prop\_expr\, {'}or{'}\, prop\_expr \,|\, prop\_expr\, {'}and{'}\, prop\_expr \,| \\
& \quad\; prop\_expr\, {'}imply{'}\, prop\_expr \\
property &\rightarrow id{'}.{'}id \,|\, clock\ rel\ cexpr \,|\, iguard \,|\, {'}deadlock{'}
\end{aligned}
$$

Where $id'.'id$ denotes a variable or a location in an automaton, $clock$ denotes a clock variable and rel, $cexpr$ and $iguard$ are defined as for guards on page 11.

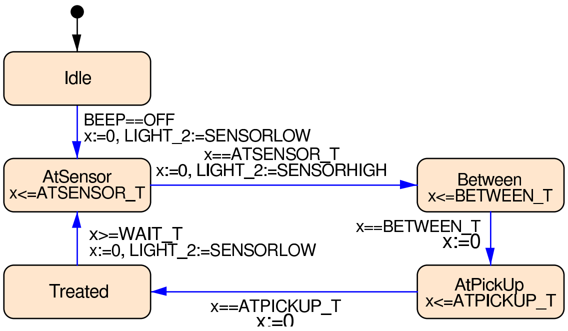


Figure 5.8: Model of a stream of sporadically arriving bricks.

The main goal of the analysis is to establish if the design is schedulable when the production cell operates to move bricks through the system. In particular we study that the cell can handle a regular stream of bricks and that it can handle a burst of bricks arriving in a shorter interval. Furthermore we check for simpler properties that tests if the design model behaves as expected, for example that a brick can be picked up.

5.2.1 Environment models

The relevant objects in the environment of the production cell controller are the bricks arriving on the feed belt and the robot.

Modelling bricks on the feed belt There are two different abstract models of bricks. The first describes periodically arriving bricks, the second describes a burst of bricks.

The first BRICK model is shown in Figure 5.8 where the automaton contains a loop to model a stream of bricks arriving sporadically. The brick model interacts with the controller system via the sensor value `LIGHT_2`. On the edges to AtSensor the sensor value is set to the constant `SENSORLOW`, which models that the brick blocks out the light sensor. On the edge to Between `LIGHT_2` is again set to the high value `SENSORHIGH`.

The clock x is used to assure that the brick stays in AtSensor for `ATSENSOR_T` ms, in Between for `BETWEEN_T` ms and in AtPickUp for `ATPICKUP_T` ms. The guard `BEEP==OFF` assures that the brick does not enter the system while it cannot be handled.

The second BRICK model is similar to the first, but does not contain a loop. Instead we create several instances of the automaton shown in Figure 5.9, where we specify different values of the constants `earliest` and `latest` for each instance.

Modelling the robot The abstract model of the robot is shown in Figure 5.10. It interacts with the controller through the sensor variable `ROTATION_1` and the actuator variable `ROB_DIR`. The variable `ROB_DIR` corresponds to the real systems actuator variable `MOTOR_C`, but is abstracted to only take the values

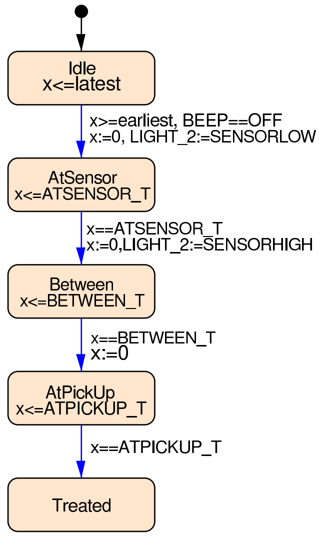


Figure 5.9: Model of a brick that moves on the feed belt.

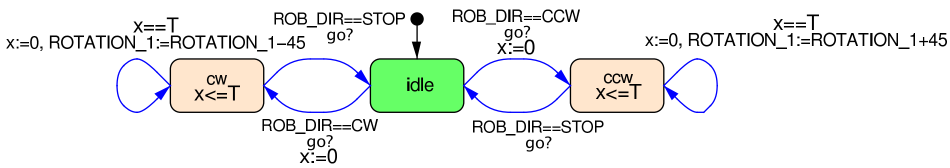


Figure 5.10: Model of the robots movement.

`CW` (clock wise), `CCW` (counter clock wise) or `STOP`. The variable `ROTATION_1` is updated in steps of 45 degrees.

Initially the robot is idle with `ROB_DIR=STOP.` The controller can start the robot by setting `ROB_DIR` to either `CW` or `CCW`. This is done in the interfaces of the tasks MvLeft and MvRight (released in automaton MoveTo). When the robot starts it enters the locations cw or ccw, where it periodically (with period T) decreases or increases the variable `ROTATION_1` by 45 degrees. The robot stops moving when `ROB_DIR` is set to `STOP` (in the assignment of task StopRobot of automaton MoveTo).

5.2.2 Verification properties

Verification of the model concerns several types of properties. One class is reachability properties that check if the system can reach states it should be able to reach. That is if the system behaves as designer intended. An example of such a property would be if the robot can be in a position to pick up a brick when there is a brick at the pickup position. This is formalised (using timed CTL) as:

$$\exists\Diamond(RobotControl.WaitAtFB \wedge Pos[0] == 0)$$

which should be read "There exists an execution where RobotControl is in location WaitAtFB and the first element in Pos is zero". Several similar reachability properties are listed in items 1 through 7 below.

With reachability analysis we are also able to determine that some parts of the model are unreachable. For example we find that the property

$$\forall\Box\neg(RobotControl.WaitP1)$$

which should be read "WaitP1 in RobotControl is not reachable", holds. This information can be used to optimise the code-generation by eliminating dead code, see Section 3.4. Properties 8 and 9 below are two such properties that where used to eliminate dead code.

Some properties are also formalisations of external requirements of the design. For example that the robot should only wait at the pick-up position of the feed belt when there is a brick approaching on the belt. This is formalised as:

$$\forall\Box(RobotControl.WaitAtFB \Rightarrow Pos[0] \geq 0)$$

Several similar properties are listed as items 10 through 13 below.

Finally the two pre-requisites for code generation, boundedness and schedulability are defined in items 15 and 16.

1.	Robot can become ready to pick up a brick from the feed belt. `E<>( RobotControl.WaitAtFB and Pos[0]==0)`
2.	Scheduler can execute the task to pick up a brick from the feed belt. `E<>( SCHEDULER.RUN_PickUpA )`
3.	Robot arm B can reach the position of the press. `E<>( RobotControl.AtPressB )`
4.	Robot arm A can reach the position of the press. `E<>( RobotControl.AtPressA )`
5.	Scheduler can execute the task to drop a brick from arm A to the press. `E<>( SCHEDULER.RUN_DropA )`
6.	Scheduler can execute the task to pick up a brick from the press with arm B. `E<>( SCHEDULER.RUN_PickUpB )`
7.	Scheduler can execute the task to drop a brick from arm B on the deposit belt. `E<>( SCHEDULER.RUN_DropB )`
8.	The robot can never reach a state where it is waiting for the press to complete (with arm B at the position of the press). Thus, location WaitP1 is dead-code in the robot controller model. `A[]not( RobotControl.WaitP1 )`
9.	Robot can never reach a state where it is waiting for press to become ready (with arm A at the position of the press). Thus, location WaitP2 is dead-code in the robot controller model. `A[]not( RobotControl.WaitP2 )`

10.	The robot controller is never waiting unnecessarily at the pick-up position of the feed belt. If it is there, it is because a brick has been detected on the feed belt (but perhaps not yet arrived to the pick-up position). `A[]( RobotControl.WaitAtFB imply Pos[0]>=0 )`
11.	When the robot controller is waiting for a brick there should be no brick on arm A. `A[] not( RobotControl.AtWait and RobotControl.PlateOnA==TT)`
12.	Whenever task PickUpA is executing, a brick is at the pick-up position of the belt. `A[]( SCHEDULER.RUN_PickUpA imply Brick.AtPickUp )`
13.	The position of the robot is always in the interval [FB_POS, DB_POS]. `A[]( ROTATION_1>=FB_POS and ROTATION_1<=DB_POS )`
14.	It should be an error if the first position of Pos is empty while numBricks indicates that there are bricks in the system. `A[] not( Pos[0]==-1 and numBricks>=1 )`
15.	Boundedness Analysis: The number of simultaneously released tasks are bounded to 3. `A[](( SCHEDULER.n0 + SCHEDULER.n1 + SCHEDULER.n2 + SCHEDULER.n3 + SCHEDULER.n4 + SCHEDULER.n5 + SCHEDULER.n6 + SCHEDULER.n7 + SCHEDULER.n8 + SCHEDULER.n9 + SCHEDULER.n10 ) <= 3 )`
16.	Schedulability Analysis: all tasks are always guaranteed to meet their deadlines. `A[]not( SCHEDULER.ERROR )`

5.2.3 Verification results

The system has been verified using a machine equipped with two 1.8 GHz AMD processors[2] and 2 GB of main memory running Mandrake Linux with kernel version 2.4.21. When we use the first brick model shown in Figure 5.8 the most time and space consuming properties needs 60 MB of main memory and takes 11 minutes to verify. With the second brick model shown in Figure 5.9 the resources used by the verification varies with the number of bricks and their values of the parameters `earliest` and `latest`. With for example three bricks with parameters as in Table 5.4 the verification consumes 14 MB of main memory and takes 14 seconds to verify.

Information from the analysis have been used to improve the design in several ways. The reachability properties 1 to 7 where helpful in the initial development where they helped to find typos and design errors in the controller automaton. Later the schedulability analysis helped to identify that the deadlines of the tasks DropA, DropB, PickUpB, UpdatePos and RdAngSen was initially to short. They where initially set to 10 ms but was modified to 16 ms, 13 ms, 17 ms, 17 ms and 16 ms respectively.

It was also found that the priorities of the tasks released by MoveTo did

[2]The verification server of Times is not implemented to make use of multiple processors so it only used one of them.

Table 5.4: Parameters to brick models.

Name	earliest	latest
Brick 1	0	500
Brick 2	500	1000
Brick 3	1000	2000

Table 5.5: Files generated by Times for the production cell.

File	Contains
Prodcell.c	Controller data structures and task code.
Prodcell_init.{h,c}	User supplied initialisation code for hardware.
brickos_interface.h	API to functions in brickos_kernel.c
brickos_kernel.c	Functions executing the kernel (see Procedure 1).
brickos_system.h	Definitions of types and macros used by code.
brickos_hooks.h (opt.)	Definitions of hooks in code for logging.
brickos_logging.h (opt.)	API for brickos_logging.c
brickos_logging.c (opt.)	Implementation of logging of kernel over LNP.

not preserve the intended execution order. The task RdAngSen should precede the tasks MvLeft and MvRight. The automaton MoveTo was also modified since it was found that two instances of task RdAngSen could erroneously be ready for execution simultaneously.

5.3 Code generation

We have used Times to generate C-code from the design model which was compiled and executed on a RCX control brick running brickOS. The code generated from the analysed model (where the dead code has been eliminated) can be found in Listing A.1 of Appendix A. Together with the encoding of the control structure Times produces a set of system files, listed in Table 5.5. Thees files contains the implementation of Procedure 1 and can be found in Listing A.2. Accompanying the kernel-file there are two files, brickos_interface.h and brickos_system.h, that contains API definitions and type declarations for the kernel respectively.

The implementation of the controller procedure in brickos_kernel.c follows the description in Section 3.2. A minor difference is that there is no separate data structure ACTIVE. Instead the table of edges include an extra boolean field `active` to indicate if the edge is among the active ones.

Optionally the code can be generated with logging capabilities. Then events in the kernel, such as “transition taken” and “task started” are logged and sent to a stationary computer using infra red communication. The operating system brickOS implements a simple communication protocol LNP (LegOS Network Protocol) using the infra red transceiver. The current implementation supports the events: “transition taken”, “task released”, “task started” and “task finished”. The files brickos_logging.h, brickos_logging.c and brickos_hooks.h contains the implementation of the logging.

Chapter 6

Conclusions and Future work

In this licentiate thesis we have studied the executable behaviours of timed systems modelled as timed automata with tasks. Our main focus has been to develop an automatic code generator for such models that produces code with predictable timing and resource utilisation.

The code generator uses a compilation procedure to transform design models to executable code, including a run-time scheduler, preserving the correctness and schedulability of the model. The code generator can also transform environment models to executable code that can be used in an animator to graphically illustrate the behaviour of the controller in a simulated environment. We interpret the environment models as hybrid automata by restricting the associated tasks to define differential equations that continuously update real-valued variables. The animator uses the differential equations solver CVODE to calculate the values of the real-valued variables at discrete time points.

As a prerequisite for code generation we require that design has been analysed to be schedulable. Other analysis results can be used to further optimise the code e.g. by limiting the amount of allocated memory or to exclude code segments that are guaranteed to be unreachable. The code generator can then, based on the deterministic semantics, transform the analysed model to executable code. Given that the target platform guarantees the synchronous hypothesis and that the tasks consumes their given computation times the code will preserve timing, resource, and logical constraints imposed on the design.

We believe that the pre-implementation analysis and simulation are valuable in the development process since they can give early indications of problems with the design. The analysis will also help in correcting the problem by providing diagnostic traces that are useful to understand the problem. It is well known that the later in the development process a problem is found the more, in terms of effort and money, it costs to correct. So early identification and correction of problems of the design will in the end lead to faster software development.

In general a potential problem with graphical languages such as TAT is that the design description tend to become big and hard to overview. Here we believe that the notion of tasks in TAT can help. By encapsulating computations in the tasks the designer can focus on the control structure of the program. Details of for example how to interact with the hardware can be separated from the control and thereby making the design cleaner.

We have demonstrated the applicability of TAT as a design language and the use of the code generator to develop the controller software for a model of an industrial production cell. We have designed a controller for the production cell using TAT, verified safety properties of the design and produced code that have been executed to control the production cell.

6.1 Future work

During the work with this licentiate thesis several ideas have been proposed that could be investigate further. Here we mention some of them.

Language While developing the controller for the production cell we found the TAT language to be sufficient for our needs. This does not mean that it will be expressive enough for larger projects. The current TAT language only supports simple data structures and expressions in the assignments. At least two directions for extensions are possible.

Either the graphical language, based on the finite state machine structure, could be extended. For example, with hierarchies of state machines as in for example UML statecharts, with more complex data-structures such as lists, records and queues, or with more powerful expressions, possibly including function calls, as assignments. This would make it easier to express more complex computations and to encapsulate an hide subsystems of the design. The problem would be to design the extensions so that the models are still possible to analyse efficiently.

Another possibility is to develop an alternative textual language. The intention would be to give the power user a faster way to describe a design. It should also be possible to describe task code and controller structure in the same textual way.

Run-time kernel The current code generation uses an existing real-time operating systems on the target platform. While this is practical, since the generated code can use hardware drivers and the run-time system of the OS, it may not be optimal for performance and memory usage. In particular an existing OS provides a scheduler that is not necessarily optimal for our needs. The precise control of start and stop times of tasks that is assumed by our code generation forces us to use semaphores or similar constructs to send signals between the controller and the task threads.

To avoid this we plan to develop a run-time kernel tailored for our setting that would include a scheduler that directly controls the task threads.

Extended task model The task model used in this licentiate thesis assumes rather independent tasks. Times actually supports a more general task model with *precedence constraints* and *mutual exclusion* . With precedence constraints it is possible to control the order in which released tasks are executed in for example a situation where one task produces a result needed by another. With mutual exclusion it is possible to control the access to a shared resource.

Both these extensions can be handled by the schedulability analysis in Times [AFM+03] but is not yet fully implemented in the code generator.

Optimising code generation We have already discussed some simple optimisations of the code generation in Section 3.4. There are room for further optimisations especially in the area of scheduler synthesis. That is to modify the scheduler so that some criterion, such as memory or power consumption, is optimised. It would mean that the tasks are still executed so that the deadline and other constraints are fulfilled but in an order that would optimise the criterion.

Integrated WCET analysis An interesting observation is that we have good knowledge of the input variables to the tasks in a TAT design. This can be used to improve the WCET analysis of the task. One of the problems in WCET analysis is that the execution time depends on the input data and since this is seldom known only pessimistic estimates are possible. For those tasks in a TAT design that only depends on the input data (and not on the environment) we can extract the exact state of the system when the tasks are released. This means that the input to the task is known and that the WCET estimation can be improved.

Distributed TAT We have assumed a single execution unit for the TAT designs. One interesting avenue of work is towards distributed TAT execution. The problems needed to be solved would include allocation of controller automata and task executions to nodes, and communication protocols and remote synchronisations between nodes.

References

[ABB+01] 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. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modelling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Sience, pages 100–125. Springer–Verlag, 2001.

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

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES - a tool for modelling and implementation of embedded systems. In J.-P. Katoen and P. Stevens (Eds.), editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Sience*, page 460ff. Springer–Verlag, 2002.

[AFM+03] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, September 2003.

[Ant00] Panos J. Antsaklis. Special issue on hybrid systems: Theory and applications a brief introduction to the theory and applications of hybrid systems. Proceedings of the IEEE, July 2000.

[BCG+99] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki. Synthesis of software programs for embedded control applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):834–849, June 1999.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427, pages 546–550. Springer–Verlag, 1998.

[Ben02] Johan Bengtsson. *Clocks, DBMs and States in Timed Systems*. PhD thesis, Uppsala Universitet, 2002. Acta Universitatis Upsaliensis nr 39.

[Ber] Gérard Berry. *The Esterel v5 Language Primer, version 5.91*. www.esterel-technologies.org.

[Ber99] Gérard Berry. The constructive semantics of pure esterel, draft version 3. www.esterel-technologies.org, July 1999.

[BLL+96] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL: a tool-suite for automatic verification of real-time systems. In *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Sience*, pages 232–243. Springer–Verlag, 1996.

[BML99] S. Bhattacharyya, P. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2):151–166, June 1999.

[Bri03] BrickOS Website. http://brickos.sourceforge.net, Aug 2003.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[CDO97] Claude Castelluccia, Walid Dabbous, and Sean O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Transactions on Networking*, 5(4):514–524, 1997.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CH96] S. Cohen and A. Hindmarsh. CVODE, a stiff/nonstiff ODE solver in C. *Computers in Physics*, 2(10):138–43, March-April 1996.

[CWA+96] Edmund M. Clarke, Jeannette M. Wing, Rajeev Alur, Rance Cleaveland, David Dill, Allen Emerson, Stephen Garland, Steven German, John Guttag, Anthony Hall, Thomas Henzinger, Gerard Holzmann, Cliff Jones, Robert Kurshan, Nancy Leveson, Kenneth McMillan, J. Moore, Doron Peled, Amir Pnueli, John Rushby, Natarajan Shankar, Joseph Sifakis, Prasad Sistla, Bernhard Steffen, Pierre Wolper, Jim Woodcock, and Pamela Zave. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, volume 1066, pages 208–219, Rutgers University, New Brunswick, NJ, USA, 22–25 October 1995. Springer.

[ELK89] E.M. Clarke, D. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 353–362, 1989.

[EWY99] Christer Ericsson, Anders Wall, and Wang Yi. Timed automata as task models for event-driven systems. In *Proceedings of the* 6th *International Conference on Real-Time Computing Systems and Applications*. IEEE, 1999.

[FMPY03] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis using two clocks. In *Proc. of the* 9th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[FPY02] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In J.-P. Katoen and P. Stevens (Eds.), editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Sience*, page 67ff. Springer–Verlag, 2002.

[Hen96] 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. IEEE Computer Society Press, 1996.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[Hol91] Gerard J. Holzmann. *Design And Validation of Computer Protocols*. Prentice Hall, 1991.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[IAR03] IAR visualSTATE website. www.iar.com/Products/VS/, March 2003.

[IL03] I-Logix. Rhapsody. www.ilogix.com/products/rhapsody/, Aug 2003.

[ITU02] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*, August 2002.

[JEK+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[LL95] Claus Lewerentz and Thomas Lindner, editors. *Formal Development of Reactive Systems: Case Study Production Cell*, number 891 in Lecture Notes in Computer Sience. Springer–Verlag, 1995.

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[LP99] Johan Lilius and Ivan Porres Paltor. Formalising uml state machines for model checking. In *Proceedings of UML'99*, volume 1723, pages 430–445, Berlin, 1999. Springer–Verlag.

[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.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

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

[Tel03] Telelogic TAU website. www.telelogic.com/products/tau/sdl/, March 2003.

[Yi91] Wang Yi. *A Calculus of Real Time Systems*. PhD thesis, Department of Computer Science, Chalmers University of Technology, 1991.

[Yov97] Sergio Yovine. KRONOS: A verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1:123–133, October 1997.

Appendix A

Code Listings

Listing A.1: Code for Producion Cell

```
/**
 * @file ProdCell.c
 * @author TimesTool, Version 1.0 beta
      , April, 2003 (build 1971) <www.
      timestool.com>
 * @brief Generated by: tobiasa
 * @date Mon Apr 14 13:38:19 CEST 2003
 * - with Target:          brickos
 */

#define LNP_LOGGING
#define HOOK_ENABLED

#include "ProdCell.h"
#include "brickos_interface.h"
#include "ProdCell_init.h"

/**
 * Task identifiers (tid).
 */
#define tid_offset 200
#define tid_AtStart tid_offset+0
#define tid_Calib tid_offset+1
#define tid_StopRobot tid_offset+2
#define tid_RdAngSen tid_offset+3
#define tid_MvRight tid_offset+4
#define tid_MvLeft tid_offset+5
#define tid_UpdatePos tid_offset+6
#define tid_PickUpA tid_offset+7
#define tid_PickUpB tid_offset+8
#define tid_DropA tid_offset+9
#define tid_DropB tid_offset+10
#define tid_NOP tid_offset+11

/**
 * Transition identifiers.
 */
#define T1 0
#define T2 1
#define T3 2
#define T4 3
#define T5 4
#define T6 5
#define T7 6
#define T8 7
#define T9 8
#define T10 9
#define T11 10
#define T12 11
#define T13 12
#define T14 13
#define T15 14
#define T16 15
#define T17 16
#define T18 17
#define T19 18
#define T20 19
#define T21 20
#define T22 21
#define T23 22
#define T24 23
#define T25 24
#define T26 25
#define T27 26
#define T28 27
#define T29 28
#define T30 29
#define T31 30
#define T32 31
#define T33 32
#define T34 33
#define T35 34
#define T36 35
#define T37 36

/**
 * Location identifiers, also offsets
 * into location array.
 */
#define RobotControl_S0 0
#define RobotControl_AtW8 2
#define RobotControl_AtPressB 5
#define RobotControl_Moving2PrB 9
#define RobotControl_W8AtFB 11
#define RobotControl_Moving2FB 13
#define RobotControl_S1 15
#define RobotControl_MovingToPrA 17
#define RobotControl_AtPressA 19
#define RobotControl_S2 22
#define RobotControl_AtPress 24
#define RobotControl_Moving2DB 27
#define RobotControl_Moving2W8 29
#define RobotControl_S3 31
#define RobotControl_AtDB 33
#define RobotControl_MovingTwPr 35
#define RobotControl_Branch 37
#define MoveTo_MR 40
#define MoveTo_Read 42
#define MoveTo_Entry 46
#define MoveTo_SR 48
#define MoveTo_ML 50
```

```
#define Enter_W8FrontEdge 54
#define Enter_S0 52


#
   define Enter_S1 56
#define Enter_W8BackEdge 58
#define UpdatePositions_Idle 60
#define UpdatePositions_Active 62

char release_list[NB_TASK]=
  {0,1,0,0,0,0,0,0,0,0,0};

/**
 * Constant values
 */
#define BELT_LENGTH 10
#define BRICK_SHIFT 1
#define FB_POS 0
#define WAIT_POS 45
#define PRESS_B_POS 0
#define PRESS_A_POS 90
#define DB_POS 90
#define PREP_T 200
#define PRESS_T 250
#define STOP 0
#define CW 1
#define CCW 2
#define TT 1
#define FF 0
#define ON 1
#define OFF 0
#define BRICKS 5
#define RobotControl_TM2PRESS 1000
#define RobotControl_GOPRESS_T -750
#define MoveTo_POLL_T 30
#define MoveTo_PREC 3
#define Enter_SAFE_TIME 4000
#define Enter_DL_CALIB 0
#define Enter_SAFE_CALIB 4000
#define UpdatePositions_UpdateTime 333

/**
 * Clock variables
 */
time_t clock_RobotControl_PressTime;
time_t clock_RobotControl_PrepTime;
time_t clock_MoveTo_x;
time_t clock_Enter_ArrivalTime;
time_t clock_UpdatePositions_x;

/**
 * Integer variables
 */
int numBricks=0;
int RobotAngle=45;
int StartSen=0;
int EndSen=0;
int Pos[5]={-1,-1,-1,-1,-1};
int AlarmTaskDone=0;
int TaskDone=1;
int Goal=0;
int ROB_DIR=0;
int edge=0;
int normalLight=0;
int BEEP=0;
int RobotControl_PlateInPress=0;
int RobotControl_PlateOnA=0;
int RobotControl_PlateOnB=0;

/**
 * Channel identfiers, one each for
 * sending and receiving, also offsets
 * into chanusage[] array
 */
#define startS 0
#define stopS 11
#define goS 14
#define startR 15
#define stopR 17
#define goR 24
#define recv_channels startR

/**
 * List of transitions that uses
 * a channel.
 */
int chanusage[25] = {
  T2,T5,T6,T8,T12,T13,T17,T18,T22,T23,
       NB_TRANS
  ,T27,T30,NB_TRANS
  ,NB_TRANS
  ,T24,NB_TRANS
  ,T1,T4,T9,T14,T15,T21,NB_TRANS
  ,NB_TRANS
};

/**
 * Evaluate guards on transition trn.
 * @param trn Transition id
 * @return true if guard satisfied,
 * false otherwise.
 */
bool eval_guard(int trn) {
  switch(trn) {
  case T2: return (TaskDone==TT);
  case T3: return (Pos[0]==0);
  case T5: return (Pos[0]>-1);
  case T6: return (
       RobotControl_PlateInPress==TT&&
       Pos[0]==-1);
  case T7: return (TaskDone==TT);
  case T8: return (
       RobotControl_PlateInPress==FF&&
       Pos[0]==-1);
  case T10: return (TaskDone==TT);
  case T11: return (
       RobotControl_PlateOnA==FF);
  case T12: return (
       RobotControl_PlateOnB==TT);
  case T13: return (
       RobotControl_PlateOnB==FF);
  case T14: return (
       RobotControl_PlateOnB==FF);
  case T16: return (TaskDone==TT);
  case T18: return (Pos[0]>-1&&
       RobotControl_PlateOnA==FF);
  case T19: return (Pos[0]==-1&&rdClock
       (RobotControl_PressTime)>PRESS_T
       &&RobotControl_PlateInPress==TT);
  case T20: return (rdClock(
       RobotControl_PrepTime)>PREP_T&&
       RobotControl_PlateOnA==TT);
  case T22: return (Pos[0]==-1);
  case T23: return (Pos[0]>-1&&
       RobotControl_PlateOnA==FF);
  case T25: return (rdClock(MoveTo_x)
       ==30&&RobotAngle<Goal);
  case T26: return (rdClock(MoveTo_x)
       ==30&&RobotAngle>Goal);
  case T28: return ((ROTATION_1>Goal?
       ROTATION_1-Goal:Goal-ROTATION_1)<
       MoveTo_PREC);
  case T29: return ((ROTATION_1>Goal?
       ROTATION_1-Goal:Goal-ROTATION_1)<
       MoveTo_PREC);
```

```
  case T31: return (AlarmTaskDone==TT);
  case T30: return (rdClock(MoveTo_x)
       ==30&&RobotAngle==Goal);



case
     T32: return (rdClock(
       Enter_ArrivalTime)>=
       Enter_SAFE_CALIB);
  case T33: return (LIGHT_2<80*
       normalLight/100);
  case T34: return (LIGHT_2>90*
       normalLight/100);
  case T35: return (Pos[0]>-1);
  case T36: return (rdClock(
       UpdatePositions_x)==
       UpdatePositions_UpdateTime&&Pos
       [0]>-1);
  case T37: return (rdClock(
       UpdatePositions_x)==
       UpdatePositions_UpdateTime&&Pos
       [0]==-1);
  case T1:
  case T4:
  case T9:
  case T15:
  case T17:
  case T21:
  case T24:
  case T27:
    return true;
  }
  return false;
}

/**
 * Perform assignments on transition
     trn.
 * @param trn Transition id.
 */
void assign(int trn) {
  switch(trn) {
  case T2:
    RobotControl_PlateOnA=TT;
    Goal=PRESS_B_POS; break;
  case T3:
    TaskDone=FF; break;
  case T5:
    Goal=FB_POS; break;
  case T6:
    Goal=PRESS_B_POS; break;
  case T7:
    RobotControl_PlateInPress=FF;
    RobotControl_PlateOnB=TT;
    setClock(RobotControl_PrepTime,0);
         break;
  case T8:
    Goal=PRESS_A_POS-45; break;
  case T10:
    RobotControl_PlateInPress=TT;
    RobotControl_PlateOnA=FF;
    setClock(RobotControl_PressTime,0)
         ; break;
  case T12:
    Goal=DB_POS; break;
  case T13:
    Goal=WAIT_POS; break;
  case T14:
    Goal=WAIT_POS; break;
  case T15:
    TaskDone=FF; break;
  case T16:
    RobotControl_PlateOnB=FF; break;
  case T17:
    Goal=WAIT_POS; break;
  case T18:
    Goal=FB_POS; break;
  case T19:
    TaskDone=FF; break;
  case T20:
    TaskDone=FF; break;
  case T22:
    Goal=PRESS_A_POS; break;
  case T23:
    Goal=FB_POS; break;
  case T24:
    setClock(MoveTo_x,0); break;
  case T31:
    BEEP=ON; break;
  case T32:
    AlarmTaskDone=FF; break;
  case T34:
    setClock(Enter_ArrivalTime,0);
    BEEP=OFF; break;
  case T35:
    setClock(UpdatePositions_x,0);
         break;
  case T36:
    setClock(UpdatePositions_x,0);
         break;
  }
}

/**
 * Transition table.
 */
trans_t trans[NB_TRANS] = {
  {false,RobotControl_Moving2PrB,
       RobotControl_AtPressB,stopS},
  {false,RobotControl_S0,
       RobotControl_Moving2PrB,startR},
  {false,RobotControl_W8AtFB,
       RobotControl_S0,-1},
  {false,RobotControl_Moving2FB,
       RobotControl_W8AtFB,stopS},
  {true,RobotControl_AtW8,
       RobotControl_Moving2FB,startR},
  {true,RobotControl_AtW8,
       RobotControl_Moving2PrB,startR},
  {false,RobotControl_S1,
       RobotControl_AtPressB,-1},
  {false,RobotControl_AtPressB,
       RobotControl_MovingTwPr,startR},
  {false,RobotControl_MovingToPrA,
       RobotControl_AtPressA,stopS},
  {false,RobotControl_S2,
       RobotControl_AtPressA,-1},
  {false,RobotControl_AtPressA,
       RobotControl_AtPress,-1},
  {false,RobotControl_AtPress,
       RobotControl_Moving2DB,startR},
  {false,RobotControl_AtPress,
       RobotControl_Moving2W8,startR},
  {false,RobotControl_Moving2W8,
       RobotControl_AtW8,stopS},
  {false,RobotControl_Moving2DB,
       RobotControl_S3,stopS},
  {false,RobotControl_S3,
       RobotControl_AtDB,-1},
  {false,RobotControl_AtDB,
       RobotControl_Moving2W8,startR},
  {false,RobotControl_AtPressB,
       RobotControl_Moving2FB,startR},
  {false,RobotControl_AtPressB,
       RobotControl_S1,-1},
```

```
  {false,RobotControl_MovingTwPr,
       RobotControl_Branch,stopS},
  {false,RobotControl_AtPressA,
       RobotControl_S2,-1},



{
   false,RobotControl_Branch,
       RobotControl_MovingToPrA,startR},
  {false,RobotControl_Branch,
       RobotControl_Moving2FB,startR},
  {true,MoveTo_Entry,MoveTo_Read,startS
       },
  {false,MoveTo_Read,MoveTo_MR,-1},
  {false,MoveTo_Read,MoveTo_ML,-1},
  {false,MoveTo_SR,MoveTo_Entry,stopR},
  {false,MoveTo_MR,MoveTo_SR,-1},
  {false,MoveTo_ML,MoveTo_SR,-1},
  {false,MoveTo_Read,MoveTo_Entry,stopR
       },
  {true,Enter_S0,Enter_W8FrontEdge,-1},
  {false,Enter_S1,Enter_S0,-1},
  {false,Enter_W8FrontEdge,
       Enter_W8BackEdge,-1},
  {false,Enter_W8BackEdge,Enter_S1,-1},
  {true,UpdatePositions_Idle,
       UpdatePositions_Active,-1},
  {false,UpdatePositions_Active,
       UpdatePositions_Active,-1},
  {false,UpdatePositions_Active,
       UpdatePositions_Idle,-1}
};

/**
 * Location list
 */
loc_t loc[NB_TRANS+NB_LOC] = {
  T2,tid_PickUpA/*S0*/,
  T5,T6,tid_NOP/*AtW8*/,
  T8,T18,T19,tid_NOP/*AtPressB*/,
  T1,tid_NOP/*Moving2PrB*/,
  T3,tid_NOP/*W8AtFB*/,
  T4,tid_NOP/*Moving2FB*/,
  T7,tid_PickUpB/*S1*/,
  T9,tid_NOP/*MovingToPrA*/,
  T11,T20,tid_NOP/*AtPressA*/,
  T10,tid_DropA/*S2*/,
  T12,T13,tid_NOP/*AtPress*/,
  T15,tid_NOP/*Moving2DB*/,
  T14,tid_NOP/*Moving2W8*/,
  T16,tid_DropB/*S3*/,
  T17,tid_NOP/*AtDB*/,
  T21,tid_NOP/*MovingTwPr*/,
  T22,T23,tid_NOP/*Branch*/,
  T28,tid_MvRight/*MR*/,
  T25,T26,T30,tid_RdAngSen/*Read*/,
  T24,tid_NOP/*Entry*/,
  T27,tid_StopRobot/*SR*/,
  T29,tid_MvLeft/*ML*/,
  T31,tid_Calib/*S0*/,
  T33,tid_NOP/*W8FrontEdge*/,
  T32,tid_AtStart/*S1*/,
  T34,tid_NOP/*W8BackEdge*/,
  T35,tid_NOP/*Idle*/,
  T36,T37,tid_UpdatePos/*Active*/
};

/**
 * Task bodies
 */
int AtStart() {
  TASK_BEGIN(AtStart)
  /* Insert "brick" at first non used
       position */
  Pos[numBricks]=BELT_LENGTH;
  numBricks++;
  TASK_END(AtStart)
}

int Calib() {
  TASK_BEGIN(Calib)
  /* Take the average light value over
       500 ms*/
#define SAMPLES 10
  int i;
  int readsum = 0;
  extern int normalLight;
  for( i=0; i < SAMPLES; i++) {
    readsum += LIGHT_2;
    msleep(50);
  }

  /* Update shared variables*/
  normalLight=readsum/SAMPLES;
  AlarmTaskDone=TT;
  TASK_END(Calib)
}

int StopRobot() {
  TASK_BEGIN(StopRobot)
  /*Stop motor moving robot*/
  motor_c_dir(brake);
  TASK_END(StopRobot)
}

int RdAngSen() {
  TASK_BEGIN(RdAngSen)
  /* Read angle from the rotation
       sensor */
  RobotAngle = -ROTATION_1;
  TASK_END(RdAngSen)
}

int MvRight() {
  TASK_BEGIN(MvRight)
  /* Start robot motor moving right*/
  motor_c_dir(rev);
  motor_c_speed(100);
  TASK_END(MvRight)
}

int MvLeft() {
  TASK_BEGIN(MvLeft)
  /* Start robot motor moving left*/
  motor_c_dir(fwd);
  motor_c_speed(100);
  TASK_END(MvLeft)
}

int UpdatePos() {
  TASK_BEGIN(UpdatePos)
  /* Update the position array*/
  int i;
  for (i=0; i<BRICKS; i++) {
    Pos[i] = ((Pos[i] >= 0) ? (Pos[i
         ]-1) : -1) ;
  }
  TASK_END(UpdatePos)
}

int PickUpA() {
  TASK_BEGIN(PickUpA)
  int i;
  /* Activate the magnet on arm A*/
  motor_a_dir(fwd);
  motor_a_speed(MAX_SPEED);
```

```
  /* Shift values in the position array
       .*/
  for( i=0; i<numBricks; i++) {
    Pos[i] = Pos[i+1];
  }
  numBricks--;
  Pos[BRICKS-1] = -1;
  RobotControl_PlateOnA=true;
  TaskDone=true;
  TASK_END(PickUpA)
}

int PickUpB() {
  TASK_BEGIN(PickUpB)
  /* Activate the magnet on arm B*/
  motor_b_dir(fwd);
  motor_b_speed(MAX_SPEED);
  RobotControl_PlateOnB=true;
  TaskDone=true;
  TASK_END(PickUpB)
}

int DropA() {
  TASK_BEGIN(DropA)
  /* Deactivate the magnet on arm A*/
  motor_a_dir(off);
  RobotControl_PlateOnA=false;
  TaskDone=true;
  TASK_END(DropA)
}

int DropB() {
  TASK_BEGIN(DropB)
  /* Deactivate the magnet on arm B*/
  motor_b_dir(off);
  RobotControl_PlateOnB=false;
  TaskDone=true;
  TASK_END(DropB)
}

int main(int argc, char **argv) {

  ProdCell_licCG_init();

  execi( &AtStart, 0, NULL, 12,
       SMALL_STACK_SIZE);
  execi( &Calib, 0, NULL, 11,
       SMALL_STACK_SIZE);
  execi( &StopRobot, 0, NULL, 10,
       SMALL_STACK_SIZE);
  execi( &RdAngSen, 0, NULL, 9,
       SMALL_STACK_SIZE);
  execi( &MvRight, 0, NULL, 8,
       SMALL_STACK_SIZE);
  execi( &MvLeft, 0, NULL, 7,
       SMALL_STACK_SIZE);
  execi( &UpdatePos, 0, NULL, 5,
       SMALL_STACK_SIZE);
  execi( &PickUpA, 0, NULL, 4,
       SMALL_STACK_SIZE);
  execi( &PickUpB, 0, NULL, 3,
       SMALL_STACK_SIZE);
  execi( &DropA, 0, NULL, 2,
       SMALL_STACK_SIZE);
  execi( &DropB, 0, NULL, 1,
       SMALL_STACK_SIZE);

  /*
   * Reset clock variables
   */
  setClock(RobotControl_PressTime,0);
  setClock(RobotControl_PrepTime,0);
  setClock(MoveTo_x,0);
  setClock(Enter_ArrivalTime,0);
  setClock(UpdatePositions_x,0);

  setupLogging();
  execi( &logging, 0, NULL, PRIO_LOWEST
       +1, SMALL_STACK_SIZE);
  execi( &controller, 0, NULL,
       PRIO_HIGHEST, SMALL_STACK_SIZE);

  cputw(MAKESIG);
  return 0;
}
```

Listing A.2: Kernel code for brickOS

```
/**
 * @file   brickos_kernel.c
 * @author Tobias Amnell <tobias.
      amnell@docs.uu.se>
 * @date   Thu Mar 28 14:28:23 2002
 * @brief  The kernel functions that
      executes the controller.
 */

#include "brickos_interface.h"

extern int chanusage[];
extern trans_t trans[];
extern loc_t loc[];
extern int eval_guard(int);
extern void assign(int);

#if NB_TASK>0
extern char release_list[NB_TASK];
#endif

/**
 * Release task with id tid.
 */
#if (NB_TASK>0)
#define releaseTask( tid ) do{ \
  release_list[ tid ]++; \
} while(0)
#else
#define releaseTask( tid ) do{ } while
     (0)
#endif

/*
 * Private functions
 */
#if NB_CHAN>0
static int check_synch( unsigned char
     sync );
#endif
static void clear_and_set ( unsigned
     char sync );

/**
 * Checks if it is possible to
      synchronise on a channel.
 * @param sync Channel id, i.e. an
      index into the chanusage array.
 * @return 0 if synchronisation is not
      possible, edge id of
 * complementary edge if it is.
 */
#if NB_CHAN>0
static int check_synch( unsigned char
     sync ) {
  while( chanusage[sync] < NB_TRANS ) {
    if( (trans[chanusage[sync]].active
         ==true) &&
        eval_guard( chanusage[sync] ) )
      return chanusage[sync];
```

```
    sync++;
  }
  return false;
}
#endif

/**
 * Update the list of active edges when
       a transition is taken, and
 * release task of target location.
 * @param trn The transition taken
 */
static void clear_and_set ( unsigned
     char trn ) {
  int jj;
  /* Clear outgoing transition from
       source location */
  jj=trans[trn].from;
  do {
    trans[loc[jj]].active=0;
    jj++;
  } while(loc[jj]<tid_offset);
  /* Set outgoing transition from
       target location */
  jj=trans[trn].to;
  while ( loc[jj]<tid_offset ) {
    trans[loc[jj]].active=1;
    jj++;
  }
  /* Release task of target location */
  releaseTask(loc[jj]-tid_offset);
  TASK_RELEASED_HOOK(loc[jj]-tid_offset
       );
}

/**
 * Is a channel @a id a sending channel
      ?
 */
#define IS_SEND( id ) (id <
     recv_channels )

/**
 * Check if any of the active edges are
       enabled, and if so take
 * it. Will continue until a stable
      state (no more enabled
 * transitions) is reached.
 * @param data Unused (should be null).
 * @return false when in stable state
 */
wakeup_t check_trans( wakeup_t data ) {
  int trn, compl_trn;
  for(trn=0; trn<NB_TRANS; trn++) {
    if( trans[trn].active ) {
      if( eval_guard(trn) ) {
#if NB_CHAN>0
        if( trans[trn].sync > -1 ) {
          if( (compl_trn =
              check_synch( trans[trn].
                   sync )) ) {
            if( IS_SEND(trans[trn].sync
                 ) ) {
              assign( trn );
              assign( compl_trn );

              TRANS_TAKEN_HOOK(trans[
                   trn].to);
              TRANS_TAKEN_HOOK(trans[
                   compl_trn].to);

            } else {
              assign( compl_trn );
              assign( trn );
              TRANS_TAKEN_HOOK(trans[
                   compl_trn].to);
              TRANS_TAKEN_HOOK(trans[
                   trn].to);

            }
            clear_and_set( trn );
            clear_and_set( compl_trn );
            trn=-1;
          }
        } else {
#endif
          assign( trn );
          TRANS_TAKEN_HOOK(trans[trn].
               to);
          clear_and_set( trn );
          trn=-1;
        } } } }
  return false;
}

/**
 * Wake-up function for threads (tasks)
      .
 * @param data Unused (should be null)
 */
#if NB_TASK>0
wakeup_t task_release(wakeup_t data) {
    return release_list[data];
}
#endif

/**
 * Thread body for automata controller
 * @return 0 (allways)
 */
int controller() {
  wait_event( &check_trans, 0);
  return 0;
}
```

Listing A.3: Types and macros used in code

```
/**
 * @file   brickos_system.h
 * @author Tobias Amnell <tobias.
      amnell@docs.uu.se>
 * @date   Mon Apr 22 10:55:25 2002
 *
 * @brief  Definitions of types and
      macros used by generated code.
 *
 * Define types and macros used by the
      atomaticaly generated code
 * from TimesTool.
 */

#ifndef BRICKOS_SYSTEM_H
#define BRICKOS_SYSTEM_H

#include "brickos_hooks.h"

/** Read clock value of @a clname. */
#define rdClock(clname) (sys_time -
     clock_##clname)

/** Set clock with id @a clname  to
     value @a value. */
#define setClock(clname, value) clock_
     ##clname = sys_time-value
```

```
/** Copy clock values with id @a
     clname2 to @a clname. */
#define cpClock(clname, clname2) clock_
     ##clname = clock_##clname2

/** Transition information */
typedef struct s_TransitionType trans_t
     ;

/** Location list type */
typedef int loc_t;

/** Define bool type */
typedef int bool;

/** Transition table element */
struct s_TransitionType {
  char active;      /**< Active
       transition */
  char from;        /**< Source
       location */
  char to;          /**< Target
       location id */
  signed char sync; /**< Syncronisation
        id */
};

#ifndef true
#define true (1==1)
#endif

#ifndef false
#define false (0==1)
#endif

#define SMALL_STACK_SIZE 128

#define TASK_BEGIN( taskname ) while(
     true) {\
    wait_event( &task_release, tid_##
         taskname - tid_offset );\
    TASK_BEGIN_HOOK( tid_##taskname -
         tid_offset );\
    release_list[ tid_##taskname -
         tid_offset]--; {

#define TASK_END( taskname )
     TASK_END_HOOK( tid_##taskname -
     tid_offset );\
    }}\
    return 0;
#endif /* BRICKOS_SYSTEM_H */
```