TKK Reports in Information and Computer Science
Espoo 2010 TKK-ICS-R28

MODEL CHECKING EMBEDDED CONTROL SOFTWARE

Juho Frits

Aalto University
School of Science
and Technology

TKK Reports in Information and Computer Science
Espoo 2010 TKK-ICS-R28

MODEL CHECKING EMBEDDED CONTROL SOFTWARE

Juho Frits

Aalto University School of Science and Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Aalto-yliopiston teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:
Aalto University School of Science and Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
PO Box 15400
FI-00076 AALTO
FINLAND
URL: http://ics.tkk.fi
Tel. +358 9 470 01
Fax +358 9 470 23369
E-mail: series@ics.tkk.fi

ISBN 978-952-60-3102-6 (Print)
ISBN 978-952-60-3103-3 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)
URL: http://lib.tkk.fi/Reports/2010/isbn9789526031033.pdf

Multiprint oy
Espoo 2010

ABSTRACT: Recently, embedded systems have become more and more complicated and thus traditional testing and simulation techniques for system validation are in many cases not sufficient. Additionally, the control of several real-world systems and processes require complex timing, which is difficult to verify with testing. The time scales of different delays can vary so much that the set of different timings possible to validate with testing is usually very limited. More powerful methods are needed and one formal method that can be used to verify and validate whether a complex system meets its requirements is model checking.

The goal of this work is to evaluate the applicability of model checking for embedded control software. A general model checking methodology is given along with some central guidelines for modeling real-time control systems and, especially, the control software of those systems.

Using the model checking methodology a part of the control firmware of an Uninterruptible Power Supply (UPS) is modeled with the model checking tool UPPAAL, which uses networks of timed automata as its modeling language. Ten failure cases related to the operation of the UPS were investigated and one or several specifications were formalized from each failure case using a temporal logic called Timed Computation Tree Logic (TCTL). The model of the system was verified against the system specifications and as a result of the verification two of the specifications were found to be violated.

The results of the work indicate that model checking is a promising method for verifying and finding errors of timed software controlled embedded systems.

KEYWORDS: Model checking, embedded software, real-time

CONTENTS

List of Figures 7

List of Tables 8

List of Abbreviations 9

1 Introduction 11
- 1.1 Model Checking . . . 11
- 1.2 Work Description . . . 12
- 1.3 Outline of the Work . . . 12

2 Model Checking Software Systems 13
- 2.1 Model Checking Embedded Control Software . . . 13
- 2.2 Model Checking Operating System Device Drivers . . . 14

3 Modelling and Analyzing Real-Time Systems 15
- 3.1 Finite Automata . . . 15
- 3.2 Timed Automata . . . 16
  - 3.2.1 Formal Syntax and Semantics . . . 17
  - 3.2.2 Network of Timed Automata . . . 18
- 3.3 Temporal Logics . . . 19
  - 3.3.1 Computation Tree Logic . . . 19
  - 3.3.2 Timed Computation Tree Logic . . . 21
- 3.4 The UPPAAL Model Checker . . . 23
  - 3.4.1 Modelling language . . . 23
  - 3.4.2 Query Language . . . 25

4 Model Checking Methodology for Timed Software Controllers 26
- 4.1 Overview of Timed Software Controllers . . . 26
- 4.2 An Abstract Model of Embedded Control Systems . . . 26
  - 4.2.1 Structure of the Abstract Model . . . 26
- 4.3 Model Checking Methodology . . . 27
  - 4.3.1 Modelling System Environment . . . 28
  - 4.3.2 Modelling Control Software . . . 28
  - 4.3.3 Timing Aspects . . . 31
  - 4.3.4 Verification of Real-Time Embedded Software . . . 32

5 Case Study: Model Checking the Control Software of a UPS Device 33
- 5.1 Uninterruptible Power Supply . . . 33
- 5.2 Description of the System . . . 34
- 5.3 Description of the UPPAAL Model . . . 35
  - 5.3.1 Configuration of the System . . . 35
  - 5.3.2 Structure of the UPPAAL Model . . . 35
- 5.4 Model of the System Environment . . . 36
  - 5.4.1 Hardware Models of Switches K3 and K5 . . . 38

5.5 Model of the Control Software 38
5.5.1 Model of the Main State Machine 38
5.5.2 Modeling the Control Software of Switches K3 and K5 44
5.5.3 Modeling the Bypass 46
5.6 Verified Properties 47
5.6.1 Failure Case 1 47
5.6.2 Failure Case 2 48
5.6.3 Failure Case 3 48
5.6.4 Failure Case 4 49
5.6.5 Failure Case 5 49
5.6.6 Failure Cases 6-9 50
5.6.7 Failure Case 10 51

6 Results **52**

7 Conclusions **56**
7.1 Future Work 56

Bibliography **57**

Appendices **62**

A Global Variable Declarations of the UPPAAL Model of the UPS Device **62**

B System Declarations of the UPPAAL Model of the UPS Device **64**

LIST OF FIGURES

3.1 A finite automaton 15
3.2 A timed automaton 16
3.3 Network of two timed automata 19
3.4 An example UPPAAL model of a lamp 24
4.1 Abstract model of software controlled embedded systems . . . 27
4.2 Example of a state machine and its UPPAAL model 29
4.3 Example of modeling a for loop with UPPAAL 31
4.4 Example of a counter and its UPPAAL model using a clock . . 31
5.1 System diagram of the UPS 34
5.2 Structure of the UPPAAL model 36
5.3 UPPAAL automaton modeling the system environment 37
5.4 UPPAAL automaton modeling the hardware of K3 switch . . . 37
5.5 Main state machine 39
5.6 UPPAAL automaton modeling main state machine 39
5.7 UPPAAL automaton modeling DC Starting state 40
5.8 UPPAAL automaton modeling Inverter Starting state 41
5.9 UPPAAL automaton modeling Inverter Syncing state 41
5.10 UPPAAL automaton modeling Inverter Online state 42
5.11 UPPAAL automaton modeling emergency transfer to bypass . 43
5.12 UPPAAL automaton observing the state of the K3 switch . . . 44
5.13 UPPAAL automaton updating `K3ClosedDebounced` variable . 45
5.14 UPPAAL automaton modeling the control logic of K5 switch . 45
5.15 UPPAAL automaton modeling the bypass 47
5.16 UPPAAL automaton modeling the bypass overvoltage 47
6.1 Voltages and bit assignments in case of symmetric input overvoltage 54

LIST OF TABLES

6.1 Model checking times 52
6.2 Model checking times when all failures were enabled 53

LIST OF ABBREVIATIONS

AC	Alternating Current
API	Application Programming Interface
CTL	Computation Tree Logic
DC	Direct Current
ESML	Embedded System Modeling Language
MCESS	Model Checking of Embedded Systems Software
PC	Personal Computer
RAM	Random Access Memory
SAL	Symbolic Analysis Laboratory
SCR	Silicon Controlled Rectifier
SDV	Static Driver Verifier
TCTL	Timed Computation Tree Logic
UPS	Uninterruptible Power Supply

1 INTRODUCTION

In recent years, the deployment of embedded systems [27] has broadened in many application fields including safety-critical applications. At the same time, the complexity of embedded software applications has increased. Additionally, correct timing is usually a critical part of the functionality of embedded control software. Increased complexity and hard real-time requirements create challenges for the verification of the correct operation of the software.

Traditionally, testing has been used to find bugs in embedded software [12]. However, testing is not sufficient in all cases. Testing can be used to find errors in software but it can not be used to prove the system error free. In many industries the requirements for the dependability of software are high and thus more efficient validation methods are needed. One technique for enhancing the reliability of software is model checking [13].

1.1 MODEL CHECKING

Model checking is a formal method for verifying hardware and software designs. Traditionally testing, simulation, and deductive verification have been used to validate complex systems. The advantage of model checking compared to those methods is that formal verification explores exhaustively all possible behaviors of the system, while simulation and testing usually explore only some of the possible behaviors of the system.

The model checking process consists of three tasks, which are modeling, specification, and verification. The first step is specifying the properties the system must satisfy. The properties are given in some logical formalism such as temporal logic, which can express the required behavior of the system over time.

In modeling task a system design is converted to a formal modeling language understood by a model checking tool. The model should capture all the properties which are needed to determine the correctness of the design. On the other hand, the model should abstract away from details that only make the verification process more complicated but do not affect the correctness of the system regarding to the checked properties.

The verification phase is fully automatic. When a model of the system design and a specification of the property the design must satisfy are given to a model checking tool, it automatically conducts the reasoning process. If the property is not satisfied, model checker usually gives a counterexample execution in which the property does not hold. The error trace is useful in finding where the error occurred and whether it was caused by a modeling error, a specification error, or an error in the system design.

Model checking, however, also has some disadvantages. One of them is the state explosion problem [37], which can occur if the system has many input variables or if the system consists of many components which can make transitions in parallel. In state explosion the number of global states of the model grows exponentially to the size of the model.

1.2 WORK DESCRIPTION

The objective of this work is to evaluate the suitability of model checking to timed embedded control software. Moreover, we introduce a general methodology for applying model checking to software controlled embedded systems.

In the case study the model checking methodology is adapted to model checking of an embedded control software of an Uninterruptible Power Supply (UPS) [9, 17] device. The UPS control software represents a challenging case for the evaluation of suitability of model checking as a verification method for embedded systems.

The model checking tool used in the verification is UPPAAL, which is a state of the art model checker for timed systems. UPPAAL was chosen as the model checker because the system is very timing dependent and verifying that the system operates correctly in every possible timing sequence was considered important. Several failure cases related to the operation of the UPS are examined and the model is verified against properties extracted from the failure cases.

1.3 OUTLINE OF THE WORK

The rest of this work is organized as follows. In Chapter 2 a survey of related research of model checking software systems is presented. In Chapter 3 the theory behind the UPPAAL model checker is discussed. A model checking methodology for embedded controllers is presented in Chapter 4. The case study of model checking a control software of a Uninterruptible Power Supply (UPS) is presented in Chapter 5. In Chapter 6 the results of the model checking are discussed. Finally, the conclusions of the work are in Chapter 7.

2 MODEL CHECKING SOFTWARE SYSTEMS

Model checking has recently been applied to many software systems, such as embedded control systems [29, 30, 19, 34, 25, 36] and operating system software [7, 6, 39].

2.1 MODEL CHECKING EMBEDDED CONTROL SOFTWARE

Model checking has been used to analyze several embedded control systems. A bounded model checker from Symbolic Analysis Laboratory (SAL) tool suite [16] was successfully employed in the modeling of an interrupt-dependent altitude display task of an aircraft [19]. A technique was introduced for creating an abstract model considering the relationship between state-triggered and interrupt-triggered transitions. The model checking method uses an abstraction of grouping instructions into basic blocks, which cannot be interrupted and it does not matter if the interrupt happens during the execution of the basic block or immediately after it. The approach was found to be suitable for finding flaws from dynamic behavior of a control program.

In paper [34] a model checking method for embedded microcontrollers is presented. The model is generated from the assembly language code, which is compiled from C source code of the program. In the work, model checking is applied to programs written for the ATMEL ATmega family of microcontrollers, but the method is easily retargetable to other types of microcontrollers. Two model checking approaches are used to model check assembly code. The first approach is based on a model checking tool [mc]square [33], which uses a simulator, called Avrora [35], to build the state space step-by-step. The second approach translates the assembly language program together with a description of the microcontoller into a bytecode language of NIPS virtual machine [38]. The model checking is performed using a tool called Model Checking of Embedded Systems Software (MCESS). The approaches were evaluated by model checking two example programs and both approaches were able to find two errors related to the timing of interrupts from the programs.

In BOS project [25, 36] a movable storm surge barrier was constructed in the Nieuwe Waterweg canal to protect the low, western part of the Netherlands from storm floods. Critical parts of the automated software controlled storm surge barrier system were verified using the SPIN [23] model checker. The results of using formal methods in the BOS system were promising. Model checking revealed several errors in the design of the system and the testing phase was easier because many of the faults had already been discovered and corrected when applying formal methods.

In paper [29] a verification method for component based event driven systems using the publisher/subscriber communication pattern is introduced. The method uses the GREAT [1] graph transformation tool to map the Embedded System Modeling Language (ESML) [26] application models to UPPAAL [28] timed automata. UPPAAL is then used to verify timed properties

of event-driven systems. The model checking method is successfully used to analyze the correctness of the real-time CORBA avionics applications built upon the event-driven Boeing Bold Stroke architecture.

2.2 MODEL CHECKING OPERATING SYSTEM DEVICE DRIVERS

In the SLAM project [7] model checking is applied to static analysis of C programs. The SLAM toolkit uses program analysis, model checking, and automated deduction to statically analyze temporal safety properties of sequential C programs. The toolkit can be used to automatically check that a program correctly uses an Application Programming Interface (API) to an external library. A tool, called Static Driver Verifier (SDV) [6], for analyzing Windows device drivers is built upon the SLAM toolkit. The tool is used for verifying that a device driver properly interacts with the operating system kernel.

Predicate abstraction [20] and model checking have been applied to verification of Linux device drivers [39]. A tool called DDVERIFY is created by integrating model checkers Cadence SMV [31] and BOPPO [15] into a verification tool SATABS [14], which is based on predicate abstraction. DDVERIFY can be used to verify concurrent programs with shared memory. The performance of DDVERIFY is not yet good enough for verification of realistic size device drivers based on the benchmarks presented in the paper [39]. However, DDVERIFY is a promising tool and after some optimizations it should be suitable for automated verification of Linux device drivers.

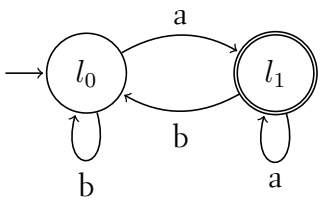

Figure 3.1: A finite automaton

3 MODELLING AND ANALYZING REAL-TIME SYSTEMS

Several model checking methods use automata as their modeling structure. Model checking can be performed using finite state automata, Büchi automata [32], timed automata [5], or some other automata formalism [11, 24].

3.1 FINITE AUTOMATA

A finite state automaton is a model of a system with a finite number of states, transitions between those states, and actions [13]. Finite automata can operate on finite or infinite words.

Definition 3.1 (Finite Automata) *A finite automaton over finite words is a five tuple* $A = \langle \Sigma, L, \Delta, L^0, F \rangle$ *such that*

- Σ *is a finite alphabet,*
- L *is a finite set of locations,*
- $\Delta \subseteq L \times \Sigma \times L$ *is the transition relation,*
- $L^0 \in L$ *is the initial location, and*
- $F \subseteq L$ *is the set of final locations.*

Automata are usually represented as graphs with labelled transitions, where the nodes represent the locations L of the automaton, and the edges are given by the transition relation Δ. The initial location is usually marked with an incoming arrow, and the final locations with double circles. In Figure 3.1, an example automaton is shown, in which $\Sigma = \{a, b\}$, $L = \{l_0, l_1\}$, $\Delta = \{(l_0, a, l_1), (l_0, b, l_0), (l_1, a, l_1), (l_1, b, l_0)\}$, $L^0 = l_0$, and $F = \{l_1\}$.

Let v be a word of Σ^* of length $|v|$. A run ρ of an automaton A over v is a mapping $\rho : \{0, 1, ..., |v|\} \mapsto L$, where:

- the first location of the run is the initial location: $\rho(0) = L^0$,
- the transition from the ith location $\rho(i)$ to the $i + 1$st location $\rho(i + 1)$ upon reading the ith input letter $v(i)$ belongs to the transition relation, that is, for each i, $0 \leq i < |v|$, $(\rho(i), v(i), \rho(i + 1)) \in \Delta$.

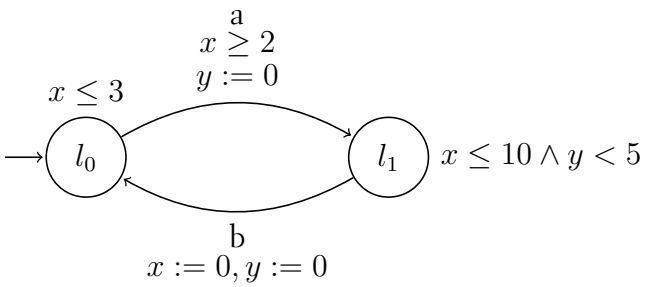

Figure 3.2: A timed automaton

A run ρ of an automaton A over v is accepting if it ends in a final location, $\rho(|v|) \in F$. A accepts a word v if and only if there exists an accepting run ρ on v. The set of accepted words of an automaton A is the language of A, $L(A)$.

3.2 TIMED AUTOMATA

A timed automaton [4, 10] is a finite state automaton extended with real-valued clock variables. All the clocks are initialized with zero when the system is started and after that they are increased at the same rate. A clock can be reset in a transition and the valuation of a clock equals to the time passed since the clock was last reset.

In the original work of Alur and Dill [4] timed automata were finite automata over infinite words extended with clock constraints, called guards, on the edges of the automata. A transition can be taken only when the values of the clocks satisfy the guard. A guard on an edge of an automaton can not force the transition to be taken, and this can lead to a situation in which the automaton stays forever in some location. This problem was solved by Alur and Dill by introducing a set of accepting locations, similar to those in Büchi automata. Only those executions passing through an accepting location infinitely often are considered valid behaviors of an automaton. By using the acceptance conditions with guard constraints an automaton can be forced to leave a location within certain time limits.

In timed safety automata [22], also a location can be associated with a clock constraint, called a location invariant. An automaton can stay in a location only as long as the invariant of the location is true. Location invariants are used instead of acceptance conditions to force an automaton to leave a location.

The structure of a timed safety automaton is simpler than that of a timed automaton, which is why timed safety automata have been adopted in many real-time verification tools such as UPPAAL [28] and Kronos [40]. In this work we focus on timed safety automata, and will refer to them as timed automata or simply automata hereafter.

An example of a timed automaton is shown in Figure 3.2. The automaton has two locations: $L = \{l_0, l_1\}$, and two clocks x and y. The starting location l_0 is marked with an incoming arrow and it has an invariant stating that the

location must be left before clock x becomes greater than 3. The invariant of l_1 is $x \leq 10 \wedge y < 5$, which means that the automaton can stay in that location as long as the clock x is less than or equal to 10 and the clock y is less than 5. The edge from l_0 to l_1 has an action a and a guard $x \geq 2$ stating that the edge can be taken when the clock x has a value greater than or equal to 2. When the edge is taken, clock y is reset. The edge from l_1 to l_0 has an action b. The edge also resets both the clocks x and y.

3.2.1 Formal Syntax and Semantics

Next the formal syntax and semantics of a timed automaton [10] are defined. To begin with, a definition of clock constraints is given. Let C be a set of non-negative real-valued clock variables. A set of clock constraints $B(C)$ is defined as follows:

- All inequalities of the form $x \sim n$ and $x - y \sim n$ are in $B(C)$, where $x, y \in C$, $n \in \mathbb{N}$, and $\sim \in \{<, \leq, =, \geq, >\}$.
- If φ_1 and φ_2 are in $B(C)$, then $\varphi_1 \wedge \varphi_2$ is in $B(C)$.

Definition 3.2 (Syntax of Timed Automata) *A timed automaton is a six tuple $A = \langle L, l_0, \Sigma, C, E, I \rangle$, where*

- *L is a finite set of locations,*
- *$l_0 \in L$ is the initial location,*
- *Σ is a finite set of actions,*
- *C is a finite set of clocks,*
- *$E \subseteq L \times \Sigma \times B(C) \times 2^C \times L$ is a set of edges between locations, each having an action, a guard, and a set of clocks to be reset,*
- *$I : L \rightarrow B(C)$ is a mapping from locations to clock constraints, called the location invariants.*

The semantics of a timed automaton is defined as a labelled transition system [21]. A clock assignment is a function $u : C \mapsto \mathbb{R}_+$ mapping the clocks to the non-negative reals. Let $\mathbb{R}^C$ denote the set of clock assignments. Let $u \models g$ denote that the clock assignment u satisfies the guard g and let $u \models I(l)$ denote that the clock assignment u satisfies all the invariant constraints of location l.

For $d \in \mathbb{R}_+$, let $u + d$ denote the clock assignment mapping all $x \in C$ to $u(x) + d$. For $r \subseteq C$, let $u' = [r \mapsto 0]u$ denote the clock assignment where for all $x \in r : u'(x) = 0$ and for all $y \in C \setminus r : u'(y) = u(y)$. Initially, all the clocks are assigned to zero value, that is, $u_0(x) = 0$ for all $x \in C$, where u_0 denotes the initial clock assignment.

A state $s = \langle l, u \rangle$ of a timed automaton consists of the current location l and the current values of the clock variables, that is, a clock assignment u. There are two types of transitions between states. In a delay transition the automaton delays for some time, that is, all the clocks of the automaton are

increased by some positive value. In an action transition an enabled edge is followed. An edge $e = \langle l, a, g, r, l' \rangle$ is enabled in a state $s = \langle l, u \rangle$ if the guard g of the edge evaluates to true and the invariant of the target state of the transition satisfies the new clock assignment obtained by resetting the set of clocks r.

Definition 3.3 (Semantics of Timed Automata) *The semantics of a timed automaton $A = \langle L, l_0, \Sigma, C, E, I \rangle$ is defined as a labelled transition system $\langle S, s_0, \Sigma, \rightarrow \rangle$, where $S = L \times \mathbb{R}^C$ is the set of states, $s_0 = \langle l_0, u_0 \rangle \in S$ is the initial state, Σ is the set of actions, and $\rightarrow \subseteq S \times \{\mathbb{R}_+ \cup \Sigma\} \times S$ is the transition relation consisting of delay and action transitions such that:*

- *for $\langle l, u \rangle \in S$ and $d \in \mathbb{R}_+$, $\langle l, u \rangle \xrightarrow{d} \langle l, u + d \rangle$ if for all $d', 0 \leq d' \leq d$: $u + d' \models I(l)$,*
- *for $\langle l, u \rangle \in S$ and $a \in \Sigma$, $\langle l, u \rangle \xrightarrow{a} \langle l', u' \rangle$ if there exists an edge $e = \langle l, a, g, r, l' \rangle \in E$ such that $u \models g$, $u' = [r \mapsto 0]u$, and $u' \models I(l')$.*

3.2.2 Network of Timed Automata

A network of timed automata [10] is a parallel composition $A_1 \parallel ... \parallel A_n$ over a common set of clocks and actions of n timed automata $A_i = \langle L_i, l_i^0, \Sigma, C, E_i, I_i \rangle$, $1 \leq i \leq n$. The action alphabet Σ consists of common actions a, used for synchronizations of the automata, and internal actions represented by a symbol τ. A location vector is a vector $\bar{l} = (l_1, ..., l_n)$. Let $I(\bar{l}) = \bigwedge_i I_i(l_i)$ be a composition of the invariant functions over the location vector. A location vector, where the ith location l_i is replaced by l_i' is denoted by $\bar{l}[l_i'/l_i]$.

Definition 3.4 (Semantics of a Network of Timed Automata)
Let $A_i = \langle L_i, l_i^0, \Sigma, C, E_i, I_i \rangle$ be a network of timed automata. The semantics is defined as a labeled transition system $\langle S, s_0, \Sigma, \rightarrow \rangle$, where $S = (L_1 \times ... \times L_n) \times \mathbb{R}^C$ is the set of states, $s_0 = \langle \bar{l}_0, u_0 \rangle \in S$ is the initial state, Σ is the set of actions, and $\rightarrow \subseteq S \times (\mathbb{R}_+ \cup \Sigma) \times S$ is the transition relation defined by the rules:

- *for $\langle \bar{l}, u \rangle \in S$ and $d \in \mathbb{R}_+$, $\langle \bar{l}, u \rangle \xrightarrow{d} \langle \bar{l}, u + d \rangle$ if for all d', $0 \leq d' \leq d$: $u + d' \models I(\bar{l})$,*
- *for $\langle \bar{l}, u \rangle \in S$, $\langle \bar{l}, u \rangle \xrightarrow{\tau} \langle \bar{l}[l_i'/l_i], u' \rangle$ if there exists an edge $e = \langle l_i, \tau, g, r, l_i' \rangle \in E_i$ for some $i \in \{1, ..., n\}$ such that $u \models g$, $u' = [r \mapsto 0]u$, and $u' \models I(\bar{l}[l_i'/l_i])$,*
- *for $\langle \bar{l}, u \rangle \in S$ and $a \in \Sigma$, $a \neq \tau$, $\langle \bar{l}, u \rangle \xrightarrow{a} \langle \bar{l}[l_i'/l_i][l_j'/l_j], u' \rangle$ if there exist edges $e_i = \langle l_i, a, g_i, r_i, l_i' \rangle \in E_i$ and $e_j = \langle l_j, a, g_j, r_j, l_j' \rangle \in E_j$, $i \neq j$, such that $u \models (g_i \wedge g_j)$, $u' = [r_i \cup r_j \mapsto 0]u$, and $u' \models I(\bar{l}[l_i'/l_i][l_j'/l_j])$.*

An example of a network of two timed automata is shown in Figure 3.3. For the initial state $\langle \bar{l}_0, u_0 \rangle$, where $\bar{l}_0 = (l_1, k_1)$ and $u_0(x) = u_0(y) = 0$, the transition relation of the network has the following transitions:

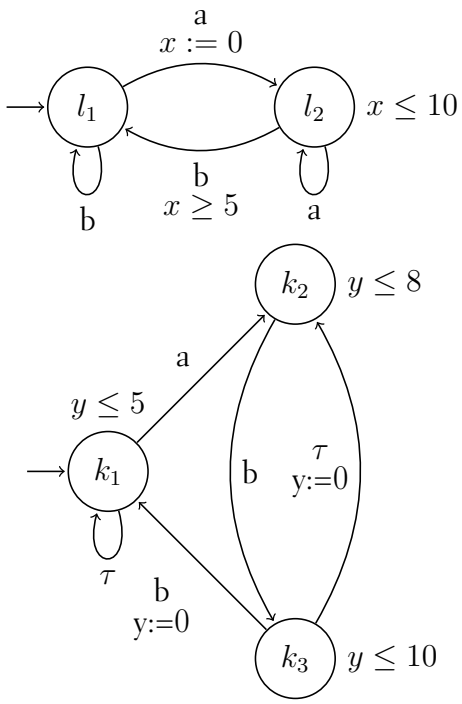

Figure 3.3: Network of two timed automata

- $\langle \bar{l}_0, u_0 \rangle \xrightarrow{d} \langle \bar{l}_0, u_0 + d \rangle$, where $d \in (0, 5]$,
- $\langle \bar{l}_0, u_0 \rangle \xrightarrow{\tau} \langle \bar{l}_0[k_1/k_1], u_0 \rangle = \langle (l_1, k_1), u_0 \rangle$, and
- $\langle \bar{l}_0, u_0 \rangle \xrightarrow{a} \langle \bar{l}_1, u_1 \rangle$, where $\bar{l}_1 = \bar{l}_0[l_2/l_1][k_2/k_1] = (l_2, k_2)$ and $u_1 = [\{x\} \cup \emptyset \mapsto 0]u_0$ meaning $u_1(x) = 0$ and $u_1(y) = u_0(y)$.

3.3 TEMPORAL LOGICS

Temporal logic is used to describe sequences of transitions between states in a reactive system. With temporal logics properties like "eventually some state is reached" or "an error state is never entered" can be specified. Temporal logics can be classified depending on the underlying time structure to linear time and branching time logics. We will focus on a branching time logic called Computation Tree Logic (CTL) [18], and its extension for real-time systems: Timed Computation Tree Logic (TCTL) [3].

3.3.1 Computation Tree Logic

CTL, introduced by Emerson and Clarke [18], is a branching-time temporal logic used as a specification language for finite-state systems. The executions of the system are modeled as linear sequences of system events. Those event sequences are called computation paths in the underlying computation tree modeling the structure of time.

CTL formulas are composed of logical operators, path quantifiers, and temporal operators. The path quantifiers are used to state if a property should

hold on some computation path (**E**) or on all computation paths (**A**) starting from the current state. The temporal operators are used to describe properties of a path through the computation tree. Five typical temporal operators are:

- **X** ("next time") requires that the property holds at the next state of the path.
- **F** ("eventually") specifies that the property holds at some state of the path.
- **G** ("globally") specifies that the property holds at every state of the path.
- **U** ("until") is a binary operator. A formula $\phi_1 \mathbf{U} \phi_2$ holds when ϕ_2 is true at some state of the path and ϕ_1 is true at every preceding state of the path.
- **R** ("release") is the logical dual of the **U** operator. The operator requires that the second argument must hold up to and including the first state where the first argument holds. The first argument is not required to become true eventually.

All CTL formulas can be defined using only **X** and **U** operators. Let AP be a set of atomic propositions. The CTL formulas are inductively defined as follows:

$$\phi := p \mid \mathbf{false} \mid \phi_1 \rightarrow \phi_2 \mid \mathbf{EX}\phi_1 \mid \mathbf{E}(\phi_1 \mathbf{U} \phi_2) \mid \mathbf{A}(\phi_1 \mathbf{U} \phi_2),$$

where $p \in AP$ and ϕ_1, ϕ_2 are CTL formulas. $\mathbf{EX}\phi_1$ means that there exists an immediate successor state in which ϕ_1 holds. $\mathbf{E}(\phi_1 \mathbf{U} \phi_2)$ specifies that there is a path having an initial prefix such that ϕ_2 holds at the last state of the prefix and ϕ_1 holds at every intermediate state. $\mathbf{A}(\phi_1 \mathbf{U} \phi_2)$ requires that the previous condition holds for every computation path. The operators **F**, **G**, and **R** can be defined using **X** and **U** operators:

- $\mathbf{EF}\phi_1 \equiv \mathbf{E}(\mathbf{true} \mathbf{U} \phi_1)$
- $\mathbf{AF}\phi_1 \equiv \mathbf{A}(\mathbf{true} \mathbf{U} \phi_1)$
- $\mathbf{EG}\phi_1 \equiv \neg \mathbf{AF} \neg \phi_1$
- $\mathbf{AG}\phi_1 \equiv \neg \mathbf{EF} \neg \phi_1$
- $\mathbf{E}(\phi_1 \mathbf{R} \phi_2) \equiv \neg \mathbf{A}(\neg \phi_1 \mathbf{U} \neg \phi_2)$
- $\mathbf{A}(\phi_1 \mathbf{R} \phi_2) \equiv \neg \mathbf{E}(\neg \phi_1 \mathbf{U} \neg \phi_2)$

The semantics of CTL formulas is defined with respect to a Kripke structure $M = \langle S, s_0, R, L \rangle$, where S is a set of states, $s_0 \in S$ is an initial state, $R \subseteq S \times S$ is a total transition relation, and $L : S \rightarrow 2^{AP}$ is a labelling function. A transition relation R is total iff for each $s \in S$ there exists a $t \in S$ such that $(s, t) \in R$.

A path in M is an infinite sequence of states $\pi = s_0, s_1, s_2, ...$ such that $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. Notation $M, s \models f$ means that f is true at state s in structure M. Let $Tr_M(s)$ be the set of paths in M starting from the state s. The satisfaction relation $\models$ is defined inductively as follows:

$$
\begin{array}{rll}
M, s \models p & \textit{iff} & p \in L(s) \\
M, s \models \neg\phi & \textit{iff} & M, s \not\models \phi \\
M, s \models \phi_1 \rightarrow \phi_2 & \textit{iff} & M, s \not\models \phi_1 \text{ or } M, s \models \phi_2 \\
M, s \models \mathbf{EX}\phi & \textit{iff} & \exists t \in S, \text{s.t. } (s,t) \in R \text{ and } M, t \models \phi \\
M, s \models \mathbf{E}(\phi_1 \mathbf{U} \phi_2) & \textit{iff} & \exists \pi = s_0, s_1, s_2, ... \in Tr_M(s) : \\
 & & \exists (i \geq 0)[M, s_i \models \phi_2 \wedge \forall (0 \leq j < i) M, s_j \models \phi_1] \\
M, s \models \mathbf{A}(\phi_1 \mathbf{U} \phi_2) & \textit{iff} & \forall \pi = s_0, s_1, s_2, ... \in Tr_M(s) : \\
 & & \exists (i \geq 0)[M, s_i \models \phi_2 \wedge \forall (0 \leq j < i) M, s_j \models \phi_1]
\end{array}
$$

The Kripke structure M satisfies ϕ iff $M, s_0 \models \phi$. A CTL formula is called satisfiable iff there exists a Kripke structure M such that $M, s \models \phi$ for some state s of M.

3.3.2 Timed Computation Tree Logic

TCTL [2, 3] is an extension to the CTL for real-time systems. In CTL it is possible to write a formula $\mathbf{EF}p$ meaning that on some computation path p will eventually become true. However, it is not possible to limit the time within which p must become true. One way to introduce real-time in temporal logic is to introduce subscripts on the temporal operators to restrict their scope in time. Using this approach to extend the syntax of CTL it is possible to write $\mathbf{EF}_{<5}p$ meaning that on some computation path p will become true within 5 time units.

We define the semantics of TCTL with respect to timed automata. To interpret the formulas of TCTL over a timed automaton, we need to extend the definition of timed automata to define which atomic propositions are true in the locations of the timed automaton. A labeled timed automaton is a pair $M = \langle A, \mu \rangle$, where $A = \langle L, l_0, \Sigma, C, E, I \rangle$ is a timed automaton and $\mu : L \rightarrow 2^{AP}$ is a labeling function assigning to each location $l \in L$ of A the set of atomic propositions true in that location. A state $s = \langle l, u, \mu(l) \rangle$ of M consists of a location l, a clock assignment u, and a set of atomic propositions $\mu(l) \subseteq AP$ true in the location l.

As opposed to CTL, which operates in discrete time domain, TCTL operates in a dense time domain. In TCTL a computation path is a map from the time domain $\mathbb{R}_+$, consisting of nonnegative reals, to the states S of the system. Along any computation path there is a unique state at every instant of time.

Definition 3.5 (s-path) *Let S be a set of states. For a state $s \in S$ an s-path through S is a map ρ from $\mathbb{R}_+$ to S satisfying $\rho(0) = s$.*

A computation tree in dense time is described by specifying a set of computation paths starting from a state $s \in S$. A prefix of an s-path ρ up to time $t \in \mathbb{R}_+$ is denoted by ρ_t. A suffix of ρ at time t, denoted by ρ^t, is a $\rho(t)$-path defined by: $\forall t' \in \mathbb{R}_+ : \rho^t(t') = \rho(t + t')$. A concatenation of $\rho' : [0, t) \rightarrow S$ and ρ, denoted by $\rho' \cdot \rho$, is defined by:

$$
\text{for } t' \in \mathbb{R}_+ : (\rho' \cdot \rho)(t') = \begin{cases} \rho'(t') & \text{if } t' < t, \\ \rho(t' - t) & \text{otherwise.} \end{cases}
$$

A map f associates a set of s-paths through S for each state $s \in S$. For example, for the initial state s_0 of a labeled timed automaton M, $f(s_0)$ gives all the possible computation paths through the states of M. A map f satisfies the following closure properties:
1. Suffix closure:

$$\forall s \in S, \forall \rho \in f(s), \forall t \in \mathbb{R}_+ : \rho^t \in f(\rho(t))$$

2. Fusion closure:

$$\forall s \in S, \forall \rho \in f(s), \forall t \in \mathbb{R}_+ : \rho_t \cdot f(\rho(t)) \subseteq f(s)$$

The fusion closure property states that the behavior of the system depends only on the current state and not on the past. Next we define the syntax of TCTL.

Definition 3.6 (Syntax of TCTL) *Let AP be a set of atomic propositions and $\mathbb{N}$ be the set of natural numbers. The formulas ϕ of TCTL are inductively defined as follows:*

$$\phi := p \mid \mathbf{false} \mid \phi_1 \rightarrow \phi_2 \mid \mathbf{E}(\phi_1 \mathbf{U}_{\sim c} \phi_2) \mid \mathbf{A}(\phi_1 \mathbf{U}_{\sim c} \phi_2),$$

where $p \in AP$, $c \in \mathbb{N}$, ϕ_1 and ϕ_2 are TCTL formulas, and $\sim \in \{<, \leq, =, \geq, >\}$.

For example, $\mathbf{E}(\phi_1 \mathbf{U}_{<c} \phi_2)$ means that there exists a computation path with an initial prefix of length less than c time units such that at the last state of the prefix ϕ_2 holds and ϕ_1 holds at every intermediate state of the prefix. Similarly, $\mathbf{A}(\phi_1 \mathbf{U}_{<c} \phi_2)$ means that every computation path has a prefix described above.

Definition 3.7 (Satisfaction relation) *Let $M = \langle A, \mu \rangle$ be a labeled timed automaton, S be the set of states of M, $s = \langle l, u, \mu(l) \rangle \in S$ be a state of M, and AP be a set of atomic propositions. The satisfaction relation $\models$ for TCTL is defined inductively as follows:*

- $M, s \models p$ *iff* $p \in \mu(l)$,
- $M, s \not\models \mathbf{false}$,
- $M, s \models (\phi_1 \rightarrow \phi_2)$ *iff* $s \not\models \phi_1$ *or* $s \models \phi_2$,
- $M, s \models \mathbf{E}(\phi_1 \mathbf{U}_{\sim c} \phi_2)$ *iff there exists a path* $\rho \in f(s)$, *for some* $t \sim c$: $\rho(t) \models \phi_2$ *and for all* $0 \leq t' < t : \rho(t') \models \phi_1$, *and*
- $M, s \models \mathbf{A}(\phi_1 \mathbf{U}_{\sim c} \phi_2)$ *iff for all paths* $\rho \in f(s)$, *for some* $t \sim c : \rho(t) \models \phi_2$ *and for all* $0 \leq t' < t : \rho(t') \models \phi_1$,

where $p \in AP$, $f(s)$ is the set of s-paths through S, and ϕ_1, ϕ_2 are TCTL formulas.

A TCTL formula ϕ is satisfiable iff there is a labeled timed automaton $M = \langle A, \mu \rangle$ and a state $s \in S$, such that $M, s \models \phi$.

As an example, take $M = \langle A, \mu \rangle$, where A is the timed automaton shown in Figure 3.2 and $\mu(l_0) = \{v\}$, $\mu(l_1) = \{w\}$. M satisfies the following TCTL formula:

$$\mathbf{E}(v\ \mathbf{U}_{<=2}\ w),$$

because there exists a path $\rho \in f(s_0)$ with an initial prefix of length 2 time units such that at the last state of the prefix w holds ($\rho(2) = \langle l_1, u_2, \{w\}\rangle$, $u_2(x) = 2, u_2(y) = 0$) and v holds in every intermediate state ($\rho(t) = \langle l_0, u_1, \{v\}\rangle, t \in [0, 2), u_1(x) = u_1(y) = t$).

3.4 THE UPPAAL MODEL CHECKER

UPPAAL [8, 28] is a real-time model checker. The modeling language of UPPAAL is based on the theory of timed automata. The query language, used for specifying properties to be checked, is a subset of TCTL. In this section we introduce the modeling and query languages of UPPAAL in detail.

3.4.1 Modelling language

The modeling language of UPPAAL is based on networks of timed automata. Templates of timed automata can be created with the graphical user interface of the UPPAAL modeling tool. Global variables and functions, accessible by every automaton, can be declared in global declaration section of the model. In addition, each automaton template has a section for local declarations of variables and functions. The automata templates are instantiated and composed as a network in a section for process declarations.

The formalism of timed automata is extended with bounded integer variables, which are part of the state. A state of the system is defined by locations of all automata, clock valuations, and the values of the integer variables. Clocks can be considered as typed variables with type *clock* and, thus, all the variables can be treated the same as clock variables in Definition 3.4. The clock assignment u in the state configuration $\langle \bar{l}, u\rangle$ can be extended to store also the values of the integer variables. The difference between clock and integer variables is that clocks can only be used to measure time and differences in time whereas integers can be used to perform more complex arithmetic operations. UPPAAL also supports arrays of clocks, constants, channels, and integer variables.

The automata, also called as processes, communicate synchronously by hand-shake synchronization using input and output actions, and asynchronously using shared integer variables. The action alphabet Σ consists of output actions $a!$, input actions $a?$, and internal actions denoted by the symbol τ. Synchronizations happen through channels. In a *binary synchronization* two automata take an action transition simultaneously with one automaton sending a synchronization (denoted with action $a!$) to a channel a, and another receiving it with action $a?$. Semantically, binary synchronization is like the third type of transition in Definition 3.4 except that the action of the sending automaton is $a!$ and the action of the receiving automaton is $a?$.

In addition to normal binary synchronization channels UPPAAL has *broadcast* and *urgent* synchronizations. In broadcast synchronization the number of receivers is not limited. The sender automaton can fire an edge with broadcast synchronization even though the receiver automata can not synchronize

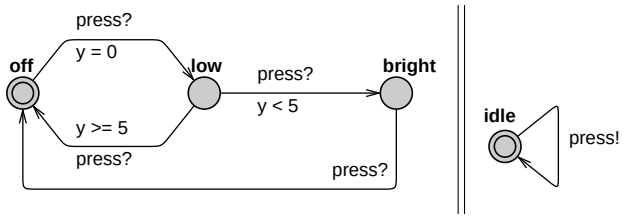

Figure 3.4: An example UPPAAL model of a lamp

in the current state. The semantics of a broadcast synchronization is like the semantics of a binary synchronization except that the number of the receiving edges is not limited and the edge with a sending synchronization can be taken even though there are not enabled edges with a receiving synchronization. However, if there are automata with enabled edges with receiving synchronization, those edges must be taken when an edge with a sending synchronization is taken.

An urgent synchronization has an exception that if a transition with urgent synchronization is enabled, the system may not delay taking it. However, interleaving with other enabled action transitions is possible even though a transition with urgent synchronization is enabled. To allow urgent synchronizations the semantics of a network of timed automata in Definition 3.4 must be changed so that delay transitions are not allowed to be taken when a transition synchronizing over an urgent channel is enabled.

Locations of UPPAAL automata can be declared *urgent* or *committed*. Time is not allowed to pass when at least one of the processes is in an urgent or committed location. Semantically, urgent locations are equivalent to adding an extra clock x that is reset on every incoming edge of the urgent location, and adding an invariant $x \leq 0$ to the location. Committed locations of UPPAAL can be used to model atomic sequences of actions. A state is committed if at least one of the locations in the state is committed. When a system is in a committed state, only transitions with at least one outgoing edge from a committed location are enabled. In the semantics of a network (Definition 3.4) delay transitions are not allowed and only those action transitions are allowed where at least one of the source locations l_i is committed when the system is in a committed state. The difference between urgent and committed locations is that all action transitions (even those not leading out of any urgent location) are allowed when one or more automata are in urgent locations.

In Figure 3.4 is an example model of a lamp and a user with a button for controlling the lamp. The model consists of two processes, which are Lamp and User. The Lamp automaton has three locations: `off`, `low`, and `bright`. The processes synchronize through channel `press`. The User automaton has only one location, named `idle`, and an edge with a sending synchronization `press!`. The edge can be taken any time.

Initially, when user presses the button, the Lamp automaton receives the synchronization and takes an edge to location `low` and the lamp is turned on. When the Lamp automaton takes the edge from `off` to `low`, a clock variable `y` is reset to zero. If the user quickly presses the button again, the lamp becomes bright (the Lamp automaton takes an edge from `low` to `bright`). Finally, the lamp is turned off if the user presses the button after the delay of

five time units (Lamp automaton still in location `low`) or when the lamp is burning brightly (Lamp automaton in location `bright`).

3.4.2 Query Language

The query language of UPPAAL [8] is a subset of TCTL. The language consists of path formulas and state formulas. A state formula is a side-effect free expression, which can be evaluated for a state independent of the behavior of the model. The expressions can be conjunctions and disjunctions over clock constraints and integer expressions. For instance, a state formula can be an expression like `c < 5 && i == 3` that is true in a state whenever clock `c` has a value less than 5 and integer variable `i` equals 3. Variables and locations of automata can be referenced using an expression on the form `A.l`, where `A` is an automaton and `l` is a variable or a location of `A`.

In the UPPAAL syntax the temporal operator **G** ("globally") is denoted as `[]` and the operator **F** ("eventually") is denoted as `<>`. The path formulas of UPPAAL's TCTL have one of the following forms:

- `A[]` ϕ — Invariantly ϕ
- `A<>` ϕ — Always eventually ϕ
- `E[]` ϕ — Potentially always ϕ
- `E<>` ϕ — Possibly ϕ
- ϕ`-->`ψ — ϕ leads to ψ

where ϕ and ψ are state formulas. The leads to operator (`-->`) is a shorthand for `A[]` $(\phi \rightarrow$ `A<>` $\psi)$.

UPPAAL's TCTL does not support restricting the time scope of the temporal operators. Instead, time constraints can be constructed by restricting the values of the clock variables in the state formulas.

For example, a property "*Is it possible for the lamp to burn brightly 3 time units after the first push of the button?*" can be formalized for the UPPAAL model in Figure 3.4 with the following TCTL formula:

```
E<> (y == 3 && Lamp.bright)
```

The formula holds in the model, because there exist an execution where the button is pressed twice within 3 time units, when the Lamp automaton is initially in location `off`.

4 MODEL CHECKING METHODOLOGY FOR TIMED SOFTWARE CONTROLLERS

4.1 OVERVIEW OF TIMED SOFTWARE CONTROLLERS

A common feature to all embedded systems is that they interact with the physical world, called environment. Typically, an embedded system receives signals from environment through sensors and sends output signals to actors, which manipulate the environment. The interaction with the physical environment happens through some specific interfaces that determine the types of signals used in the interaction.

A software controller is typically used in an embedded system to manage a dedicated task. Most control tasks include timing and those tasks are of special interest in the model checking methodology described in the following sections.

4.2 AN ABSTRACT MODEL OF EMBEDDED CONTROL SYSTEMS

In this section we present an abstract model of software controlled embedded systems for which our model checking methodology can be applied to. The model is very general in the sense that it does not make any assumptions on the devices controlled by the software. Also the structure of the software is not restricted by the model.

In software development new features are added to the software all the time. When modeling a piece of software under development, expandability of the model must be considered. It is not very efficient if the model must be created from scratch every time a new feature or modification is added to the software. The model needs to be modular and easily understandable so that it can be easily modified later on.

The purpose of the abstract model is to create a basis for modular and easily reviewable model checking models. In Section 4.3 a model checking methodology built upon the abstract model is introduced.

4.2.1 Structure of the Abstract Model

The abstract model is shown in Figure 4.1. The model consists of two parts: the controller and the system environment. The system environment is an abstraction of the controlled devices and components as well as the sensors measuring the state of the system. The system environment consists of all the parts of the physical environment affecting the state of the system. The division to controller and system environment was made because those two parts of the system can usually be modeled quite independently.

The controller models the software controller running the software under investigation. The hardware of the controller (e.g. a microprocessor) is abstracted away from the model and it is assumed to operate properly. We are only interested in verifying the correct operation of the control software. In the model of the controller there is a module for each controlled device.

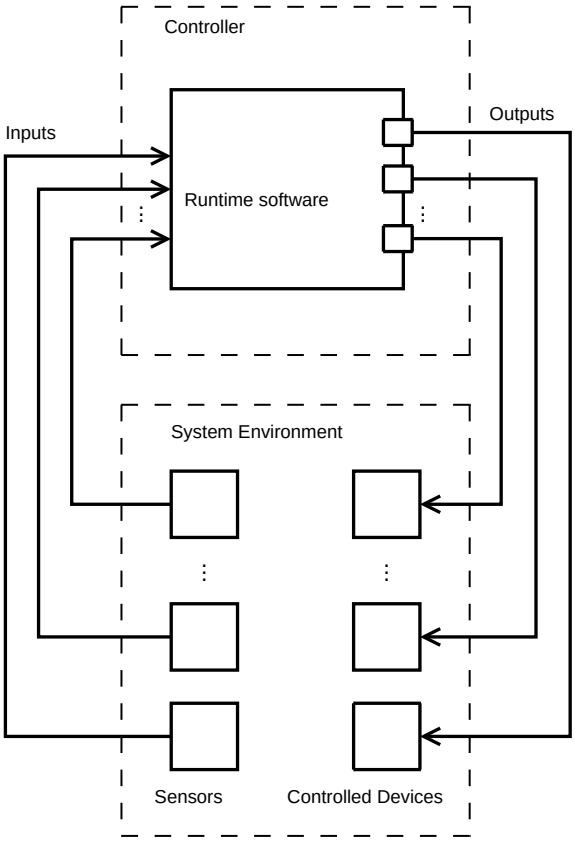

Figure 4.1: Abstract model of software controlled embedded systems

Those modules model the control logics of the software used for controlling the devices.

The arrows from the controller to the system environment represent control signals to the controlled devices. The arrows from system environment to the controller represent measurement and status signals from sensors and devices to the software controller. In this model the signals are considered to be integer valued so that they can be modeled in a straightforward way using UPPAAL's integer variables.

In the abstract model a state of the software controller is seen as a combination of the values of the internal variables and the values of the input signals. The output signals of the controller are calculated based on the current state of the controller. The state of the system environment is determined by the internal states of the controlled devices. A state of the abstract model is determined by the state of the controller and the state of the system environment.

4.3 MODEL CHECKING METHODOLOGY

In this section a methodology for modeling a software controlled real-time system using UPPAAL is described. The methodology applies to systems described in Section 4.2. The main idea is to divide the UPPAAL model in two parts: the software model and the environment model. The software model

is an abstract representation of the control software under investigation and the environment model captures the behavior of the system environment.

It is practical to begin the modeling task by determining a configuration of the system, which is to be modeled. If the system is small, it can be modeled as a whole. For bigger systems it can be impractical, or even impossible, to model the whole system. Thus, it is wise to bound the system by choosing an interesting configuration. A configuration can be bounded by, for example, assuming some variables to have a constant value all through the execution of the system. The system can also be over-approximated by allowing the values of some variables to change freely. For instance, a measurement of a process variable can be assumed to vary within certain limits and the measurement can have whatever value at any instant of time.

To reduce the state space of the model abstractions can be made by restricting the value ranges of variables. Integer and real-valued variables, such as measurements of process variables, can in many cases be abstracted to binary variables such that it is only observed if the value of the variable is higher or lower than some threshold value. These abstractions reduce the state space of the model significantly and make the modeling of bigger subsystems possible.

One of the objectives of the model checking methodology is to create a basis for modular models. In the UPPAAL modeling language there is no such concept as a module. In the following a module refers to a set of one or more timed automata operating as a whole and forming an entirety.

4.3.1 Modelling System Environment

A model of the system environment includes all the external actions of the modeled system not controllable by the software and the behavior of the controlled hardware. The system environment can be modeled, for example, by creating a module sending synchronization events when the state of the environment changes. Those events are then received by software model.

The environment model also contains modules for the hardware components, which are controlled by the software. A component model abstracts the relevant operation of the component, including the time delays related to the correct functioning of the component. If fault models are of interest, they can be embedded into the hardware models of the components.

Each hardware component controlled by the software also has a module in the software model. The module models the control logic of the software for controlling the component. The software model is in interaction with the hardware model such that the software model knows the state of the hardware the same way as the software gets status information by reading the sensors.

4.3.2 Modelling Control Software

Data structures

The only data structures UPPAAL supports are an *array* and a *struct* (similar to a structure in C programming language). If the modeled software has more complicated data structures which need to be modeled, they must be converted to simpler data structures using only those two types of structures

```
void state_machine()
{
  switch(State) {
  case STATE_0:
    state_0();
    break;

  // cases STATE_1..N-1

  case STATE_N:
    state_N();
    break;
  }
}

void state_0()
{
  // State actions

  if(timer++ >= ONE_SECOND) {
    State = STATE_1;
    timer = 0;
  }
}
```

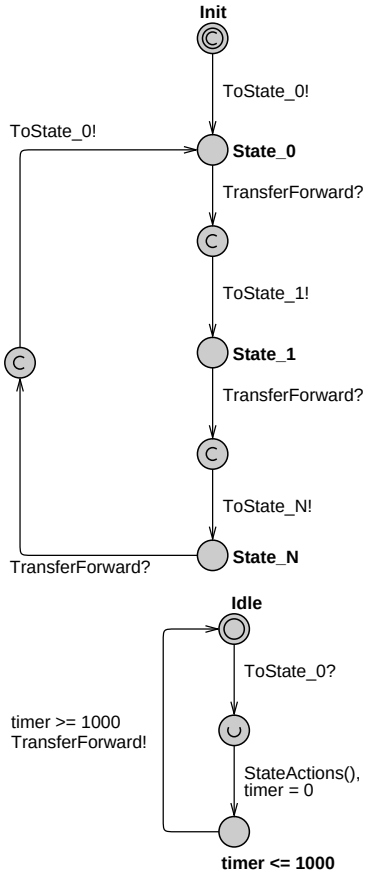

Figure 4.2: Example of a state machine and its UPPAAL model

or to several individual variables.

State machine

Many embedded software applications are implemented as state machines, which are practical if the system has several different operation modes. A state machine is easily translatable to UPPAAL modeling language by creating an UPPAAL automaton for the state machine. If the states of the state machine have several actions and complicated timing behavior associated to them, they can be modeled with one automaton per state while one automaton keeps track of the current state.

An example of a state machine written in C programming language is shown in Figure 4.2. The function `state_machine` is called periodically, for example, every 5 ms. A variable `State` keeps a tally of the current state of the state machine. Each state is implemented with a function and those functions update the value of the variable `State` when transferring to a new state. In Figure 4.2 only the implementation of state 0 is shown. In state 0 there is, in addition to some set of actions, a delay of one second before transferring to state 1. The delay is implemented with an integer-valued variable `timer`, which is incremented periodically when the state machine is in state 0. The timer timeouts after one second and the state machine transfers to state 1.

The UPPAAL model of the state machine is shown on the right side of Figure 4.2. The upper automaton models the overall structure of the state machine and the lower automaton is the model of state 0. The automata

communicate through synchronization channels. The channels `ToState_X` ($0 \leq X \leq N$) are used to inform the automata modeling individual states about entering of particular state. When leaving a state a synchronization to channel `TransferForward` is sent by the automaton modeling the state to inform the state machine automaton about state change.

In the state machine automaton committed locations are used to conjoin two synchronizing transitions together with no time passing between the transitions. If the structure of the state machine is more complicated, that is, from some state it is possible to transfer to more than one succeeding state, a global variable `NextState` can be used to tell the state machine automaton to which state the machine is transferring. A new value is assigned to `NextState` by the automata modeling the states. From the committed locations of the state machine automaton there must be edges to each of the locations modeling the successor states of each state. Those edges have guards of the form `NextState == STATE_X`.

This approach of modeling the state machine quarantees that the automata modeling the states need not know anything about each other as they only communicate with the automaton modeling the structure of the state machine. If a new state is added to the model, an automaton is created for it, which needs to implement the interface with the main state machine automaton regarding to the two channels for entering and leaving a state. This makes the model more modular as states of the state machine can be added or removed from the model fairly easily and the actions of one state are all contained within one automaton.

Control structures

The modeling of control structures is quite straightforward when using the following translations. An assignment of a new value for a variable is modeled as an update in the update section of an edge. A sequence of statements is modeled by creating several subsequent edges concatenated by committed locations. In case of a sequence of assignments, the assignments can be combined together and modeled as one edge update consisting of all the assignments. A conditional statement is modeled with two or more outgoing edges from a location with the conditions as the guards of the edges. The guards must be formulated in such a way that they evaluate to true only when the corresponding branch of the source code is executed.

A loop is modeled with an initial location having two outgoing edges. One of the edges has a guard, which evaluates to true if and only if the condition of the loop expression is fulfilled. The body of the loop is modeled as a sequence of statements and from the final location there is an edge back to the initial location of the loop. The other edge from the initial location has a guard, which evaluates to true only when the guard of the first edge evaluates to false. An example of a for loop and its UPPAAL model are shown in Figure 4.3.

A subprogram call is modeled the same way as a sequence of statements. Subprogram calls can also be modeled by using global or local functions of UPPAAL and calling the function from an appropriate edge. In functions it is possible to use for and while loops and if statements, so it is sometimes more convenient to model those control structures with functions. In gen-

```
int x, y, z;
y = 0;
z = 0;

for(x = 0; x < 5; ++x) {
  y += x;
  z = 2 * y;
}
```

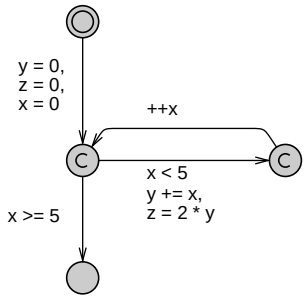

Figure 4.3: Example of modeling a for loop with UPPAAL

```
void update_counter()
{
  if(cond)
    ++c;
  else
    c = 0;
  if(c >= 100_MS) {
    x = 1;
    c = 0;
  }
}
```

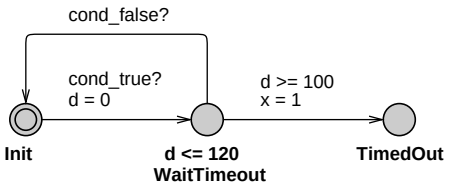

Figure 4.4: Example of a counter and its UPPAAL model using a clock

eral, keeping the number of locations of the automata as small as possible is a good idea because extra locations increase the state space of the model rapidly. Larger state space increase the time and memory consumption of the verification runs. If it is possible to model a control structure using a function, it is usually profitable to do so.

4.3.3 Timing Aspects

Correct timing is usually a critical part of a system. Many embedded systems have several time delays of varying length related to the operation and control of the system. Typically, in the source code of the control software time is handled using discrete counter variables, which are incremented periodically. When modeling counters with UPPAAL, the most practical way is to use UPPAAL's real-valued clock variables.

When modeling a discrete counter with a real-valued clock variable, it must be taken into account that the counter is incremented in intervals. For example, consider a counter `c`, which is incremented in an interrupt handling routine every 20 milliseconds when some condition `cond` is true. The counter timeouts in 100 ms, that is, when 100 ms has been elapsed from the first increment of `c`. The delay before the first increment of the counter after

`cond` becomes true can be somewhere between 0 and 20 ms. Now, assume `c` is modeled with a real-valued clock variable `d`, which is reset the exact moment `cond` becomes true. To model the behavior of `c` exactly, d must be able to timeout after 100-120 ms. The additional delay comes from the fact that `cond` may have already been true for at most 20 ms before `c` is incremented the first time.

The source code snippet of handling of the counter `c` and the UPPAAL model of it using the clock `d` are shown in Figure 4.4. The variable `cond` is modeled using two synchronization channels `cond_true` and `cond_false` to inform the automaton about `cond` changing to true or false, respectively. The edge from `WaitTimeout` to `TimedOut` can be taken when `d` has a value between 100 and 120.

4.3.4 Verification of Real-Time Embedded Software

The verified properties can typically be extracted from the interface between the controller and the system environment. In Figure 4.1 the interface is drawn with a dashed line around the controller. Usually we are interested in how the software reacts to some event from the system environment. This kind of properties can be observed from the input and output signals crossing the interface.

When making abstractions during modeling, special attention needs to be paid to the correctness of the model checking results. If the state space of the model is reduced by using over-approximation techniques, e.g., allowing the values of some variables to alternate freely, the model is an over-approximation of the system. This means that the model captures all the behaviors of the modeled system, but in addition to those there can be additional 'spurious' traces. If the model checking tool gives a positive answer, we can be sure that the system satisfies the given property (assuming the model is otherwise built properly). But if the answer is negative, the trace generated by the model checking tool may not be an execution of the actual system. In this case the model needs to be refined to get rid of the 'spurious' trace.

5 CASE STUDY: MODEL CHECKING THE CONTROL SOFTWARE OF A UPS DEVICE

In the case study the goal was to model a part of a control firmware of a UPS device and investigate whether key safety properties are satisfied in case of several failure situations such as in the case of a component failure. The model checking tool employed in the study was UPPAAL [28]. The modeling and verification of the system were carried out using the practices of the model checking methodology introduced in Chapter 4.

In Section 5.1 a general overview of different kinds of UPS designs is given. Section 5.2 gives a description of the UPS device under investigation. The UPPAAL model of the system is described in detail in Sections 5.3–5.5. Section 5.6 describes the failure cases and the properties with their TCTL formalizations regarding the operation of the UPS in those failure situations.

5.1 UNINTERRUPTIBLE POWER SUPPLY

A UPS [9, 17] provides back-up power in case of power failure and protects connected equipment from different power disturbances. The backup power is typically derived from a battery. The Direct Current (DC) of a battery is converted to Alternating Current (AC) with an inverter. The power for loading the battery is converted from AC to DC with a rectifier. Some more complicated UPS designs have an additional bypass line, which can be used if there is a failure in the primary power path.

The UPS devices can be divided to four types. The most basic type is standby power supply, which normally derives the power directly from the primary power source until power fails. In case of failure of the primary source a battery powered inverter is turned on to continue supplying power and a transfer switch switches the load over to the backup power source. The switching of power sources causes a momentary loss of power, which is unacceptable in many configurations and makes standby power supplies not suitable for those applications.

A line interactive UPS has an inverter/converter, which connects the battery continuously to the output of the UPS. Battery charging power is provided by operating the inverter in reverse when the input AC is connected. A transfer switch in the input AC line is opened when the input power fails, and battery continues to supply power to the output through inverter.

A standby-ferro UPS has a three-coil transformer connecting two power paths to the output. The primary power path comes from AC input through a transfer switch and the secondary power path from battery through an inverter.

An On-Line UPS, also called Double Conversion UPS, is the most common design in larger, above 10 kVA systems. The primary power path is from AC input through rectifier and inverter to the output. The secondary path is through a bypass line and a static bypass switch. Battery is located between rectifier and inverter, and provides power to the output in case the AC input fails. In some models the battery is continually connected to the inverter,

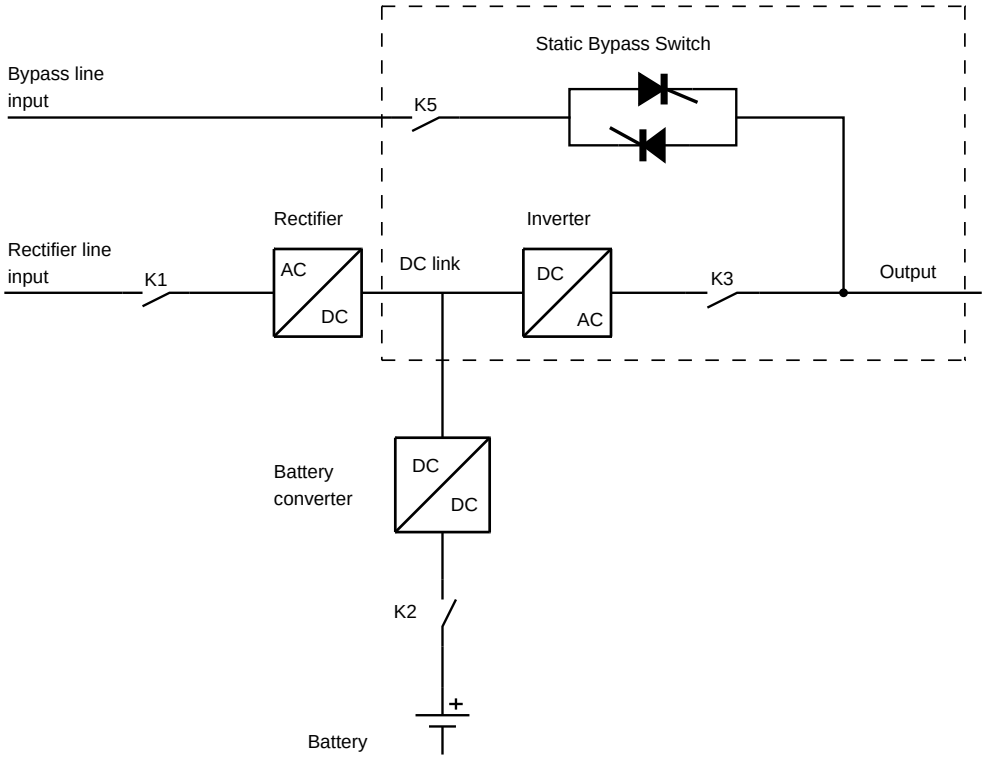

Figure 5.1: System diagram of the UPS

which means there is no transfer time during input power failure.

5.2 DESCRIPTION OF THE SYSTEM

The block diagram of the double conversion UPS investigated in this case study is shown in Figure 5.1. The UPS has two AC power connections. The primary power path is from AC input, through a rectifier, through an inverter to the output. Between rectifier and inverter is a DC link, which can also be powered from battery through a battery converter. The secondary power connection is the bypass line, which is used in different power failure situations or when maintaining the UPS device. The bypass line can be powered from the same supply source as the rectifier or from a separate supply source.

The different power paths are controlled using four switches. In the input line there is K1, which is normally closed and opened only when there is power disturbances. The power to and from battery is fed through K2. After the inverter there is the K3 switch, and in the bypass line there is a backfeed contactor called K5. The bypass line also has a static switch consisting of two Silicon Controlled Rectifiers (SCRs).

The UPS also has several measurements not presented in Figure 5.1. The switches K1, K2, K3, and K5 have auxiliary switches sensing the state of the switch. Voltage is measured from the rectifier and bypass input lines, battery, DC link, inverter output, after K5 switch, and UPS output. Current is measured from the battery output, DC link, inverter output, and bypass line.

In the case study only a part of the control firmware of the UPS was mod-

eled and verified. The subsystem (outlined with a dashed line in Figure 5.1) consists of the DC link, inverter, bypass, switches K3 and K5, and the static bypass switch. No assumptions about the environment is made. Voltages in DC link and bypass input line can change freely. Also the load can change freely as time evolves.

The main operational criterion of a UPS is to protect the UPS hardware and the load from damages while trying to maintain power at output as long as possible. Primarily, the UPS feeds power to the output through the inverter, but in case of, for example, output undervoltage the UPS transfers to bypass if it is available. In an extreme case of bypass being not available the load might be dropped when a critical alarm occurs. Most of the measurements and alarm signals have filtering and therefore there is a short delay before the UPS takes an action. The delay varies from milliseconds to several minutes depending on the criticality of the alarm.

The UPS modules can be configured to operate in parallel to increase the availability or the capacity of the UPS system. In this work we focus on the operation of a single unit.

5.3 DESCRIPTION OF THE UPPAAL MODEL

5.3.1 Configuration of the System

The modeled system is quite large and complicated and thus some restrictions are needed to be made when modeling the system. As previously mentioned, we are only interested in the operation of one unit and not in the parallel operation of several units. During the modeling phase all variables related to the parallel operation were assumed to have zero value. Thus, the firmware parts for parallel operation were not modeled at all.

The UPS device has several test modes used to analyze the operating condition of the hardware. For example, there is a test mode to test the voltage and charge levels of a battery. All those test modes were considered to be disabled when building the model.

Many of the verified properties relate to the fault models of the UPS device. However, we are only interested in verifying that the UPS is single-failure tolerant. If multiple failures can happen simultaneously, many of the properties are violated although they hold when only one failure is allowed to happen. This is why there are in the model three configuration bits for enabling or disabling failures. Those bits are: `K3_CAN_FAIL` (for enabling the failures of K3 switch), `K5_CAN_FAIL` (for enabling the failures of K5 switch), and `ENABLE_BYP_ACOV` (for enabling overvoltage in the bypass input line).

5.3.2 Structure of the UPPAAL Model

The model of the UPS system was constructed using the model checking methodology presented in Section 4.3. The structure of the UPPAAL model is shown in Figure 5.2. The structure follows the division of the abstract model presented in Section 4.2. The circles in Figure 5.2 represent the timed automata of the UPPAAL model. Dashed line is used to illustrate groups of

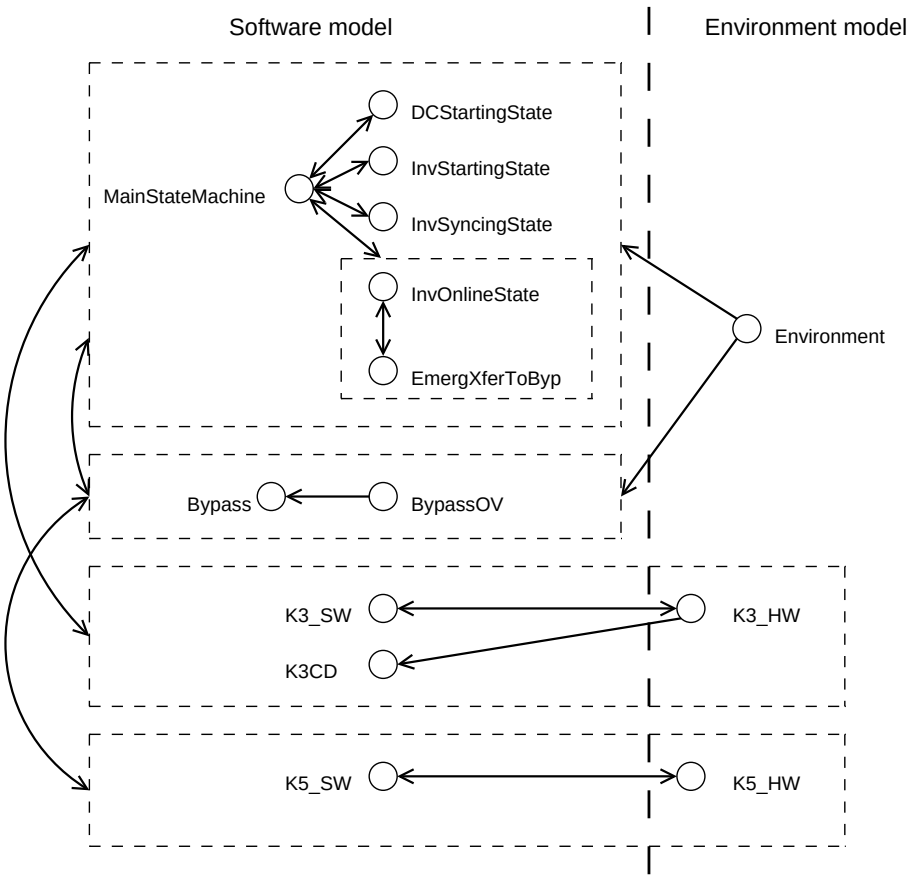

Figure 5.2: Structure of the UPPAAL model

automata, i.e., automata modeling some part of the system and operating in collaboration. The arrows between (groups of) automata illustrate the communication between the automata.

As shown in Figure 5.2, the model is divided in two parts: the software model and the model of the system environment. The model of the system environment is explained in Section 5.4 and the model of the control software is explained in Section 5.5. The declarations of global variables of the model are in Appendix A and the system declarations including process instantiations and the parallel composition of processes forming the system are in Appendix B. All the clock variables are declared locally in the local declarations sections of the automata templates.

5.4 MODEL OF THE SYSTEM ENVIRONMENT

The model of the system environment is built based on design documents of the UPS device. The environment model consists of three automata: `Environment`, `K3_HW`, and `K5_HW`. The `Environment` automaton models external events of the UPS system. The automata `K3_HW` and `K5_HW` model the hardware of the switches K3 and K5.

The `Environment` automaton sends synchronization events to the automata in software model in case of some external events. The automaton

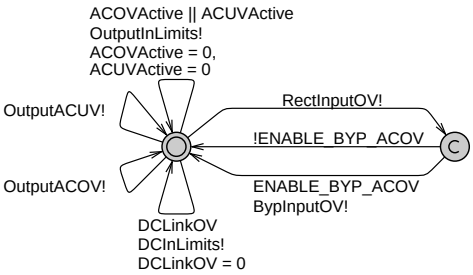

Figure 5.3: UPPAAL automaton modeling the system environment

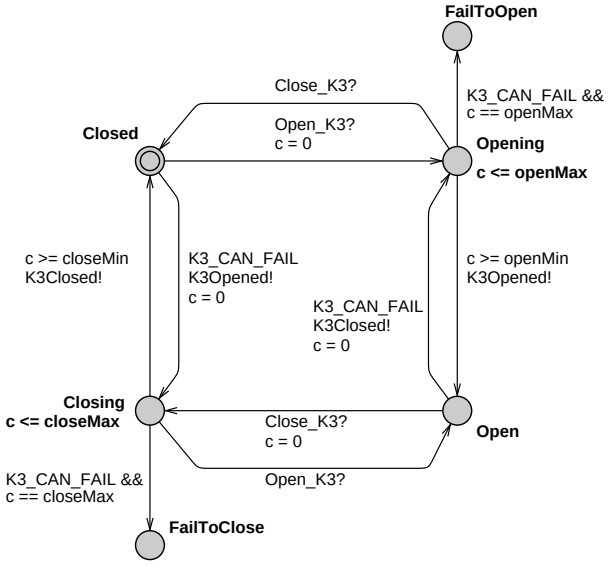

Figure 5.4: UPPAAL automaton modeling the hardware of K3 switch

consists of two locations. Initially, the automaton is in the leftmost location and from there it can take any of the five outgoing edges anytime. Each of those edges have a sending synchronization modeling a change of voltage level in UPS input or output or in the DC link.

If there is overvoltage in the input line of the rectifier, a synchronization `RectInputOV` is sent and a committed location is reached. From there, one of the two outgoing edges is fired depending on whether the configuration bit `ENABLE_BYP_ACOV` is set or not. If the bit is set (meaning there can be overvoltage in the bypass line and the bypass and rectifier inputs are connected to a common supply source), a synchronization `BypInputOV` is sent. In case of over- or undervoltage in the UPS output, a synchronization is sent to channel `OutputACOV` or `OutputACUV`, respectively. `OutputInLimits` synchronization models the normal voltage level of the UPS output. It can be sent only when an output under- or overvoltage alarm is active. Similarly, if the DC link overvoltage alarm is active, the `DCInLimits` synchronization can be sent to inform the software model about normal voltage level in the DC link.

5.4.1 Hardware Models of Switches K3 and K5

A timed automaton modeling the hardware of the K3 switch is shown in Figure 5.4. The automaton template has four integer parameters: `closeMin`, `closeMax`, `openMin`, and `openMax`. The parameters are used for specifying the opening and closing times of the switch. The times are parameters so that they can be easily changed if different types of switches are modeled.

The locations `Open` and `Closed` model when the switch is open or closed. Two intermediate locations, `Opening` and `Closing`, model the transitions between the open and closed states of the switch. For example, in the `Opening` location the switch is commanded to open but has not yet had time to open and is still closed.

A configuration bit `K3_CAN_FAIL` is used to set fault models of the switch active or inactive. When the fault models are active, two additional locations, `FailToOpen` and `FailToClose`, are reachable. Those locations model the situations when the switch will not open or close within certain time limits. When the fault models are active, the switch can also open or close spuriously. In the automaton this is modeled by the edges from the location `Open` to the location `Opening` and from the location `Closed` to the location `Closing`.

The automaton template modeling the hardware of the K5 switch is otherwise the same as the one for K3 switch shown in Figure 5.4 but the variable and channel names has been changed from K3 to K5. Those two automata could not be modeled with one template because they use different types of synchronization channels.

5.5 MODEL OF THE CONTROL SOFTWARE

The software model is built based on the control firmware of the UPS. As seen from Figure 5.2, the software model consists of a model of the main state machine, a module modeling the control of bypass, and software models for the switches K3 and K5. In following, the automata modeling the control software are introduced.

5.5.1 Model of the Main State Machine

The main state machine of the software is modeled using the principle explained in Section 4.3.2. The UPPAAL model consists of one automaton modeling the structure of the main state machine, and one automaton for each of the states DC Starting, Inverter Starting, and Inverter Syncing. The Inverter Online state is modeled with two automata because that state consists of several timed actions, which are easier and more practical to model using two automata instead of only one.

The automaton modeling the structure of the main state machine, shown in Figure 5.6, has a location for each machine state. Locations `InitializingState` and `ShutdownState` are marked urgent because those two states have not been modeled. The automaton communicates with the automata modeling the states through synchronization channels and a global variable

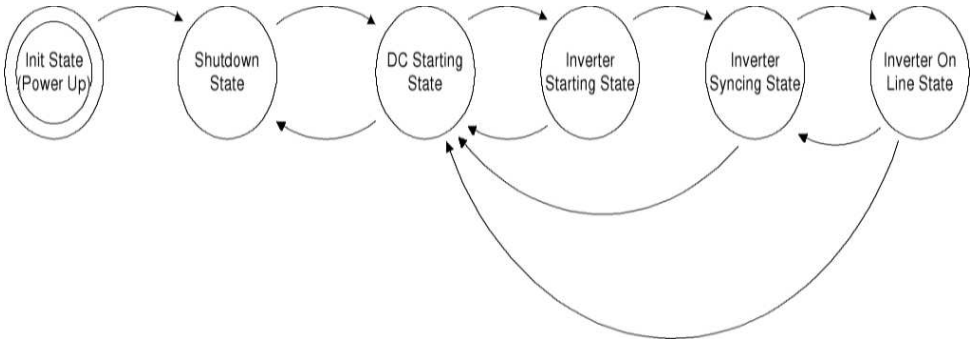

Figure 5.5: Main state machine

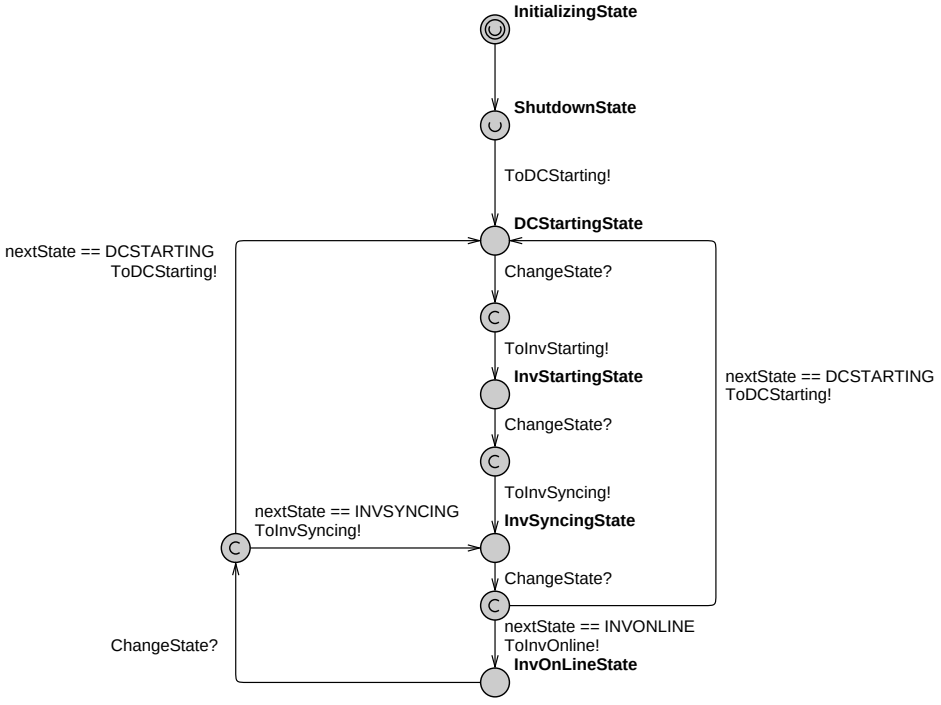

Figure 5.6: UPPAAL automaton modeling main state machine

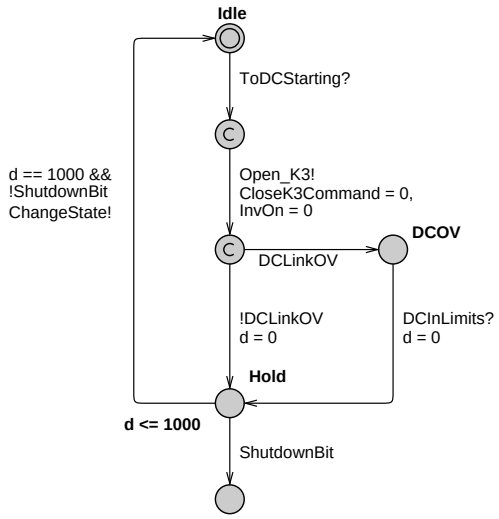

Figure 5.7: UPPAAL automaton modeling DC Starting state

`nextState` indicating the succeeding machine state. When transferring to a new state, the automaton sends a synchronization event to a channel named according to the new state. Respectively, when leaving a state the automaton modeling the state sends a synchronization event to a channel `ChangeState` to inform the main state machine automaton about state change. If it is possible to transfer to several states from some state, a variable `nextState` is assigned by a state automaton to a value corresponding the new state. That variable is used in the guard constraints of the main state machine automaton to force the automaton to take the corresponding edge leading to the desired location.

DC Starting State

In Figure 5.7 is an automaton modeling the DC Starting state. The automaton is always in a location `Idle` when the main state machine is not in the DC Starting state. When a synchronization from a channel `ToDCStarting` is received, the automaton performs the actions of the DCStarting state and after that transfers back to location `Idle`.

Immediately after transferring to the DCStarting state the automaton sends a synchronization to open the K3 switch. A variable `InvOn` is assigned to value 0 to represent the situation of inverter being turned off. After the transition a committed location is reached, from where one of the two outgoing edges is taken depending on the value of `DCLinkOV` variable. If the variable has a value of 0 (meaning there is no overvoltage present in the DC link), the automaton transfers to location `Hold`. If the `DCLinkOV` variable has a value of 1, the automaton transfers to a location `DCOV`, where it waits for a synchronization from a channel `DCInLimits` indicating that there no longer is overvoltage present in the DC link. In location `Hold` the automaton stays for one second (1000 time units) before transferring back to location `Idle`. However, if the variable `ShutdownBit` has value 1, the automaton can only fire an edge to another location from where there are no outgoing edges. `ShutdownBit` is used to prevent the main state machine from proceeding to

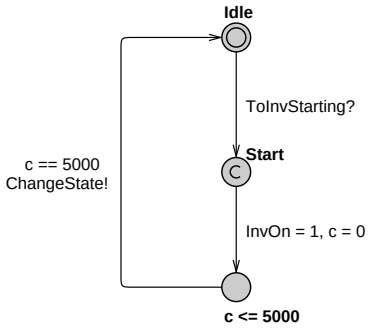

Figure 5.8: UPPAAL automaton modeling Inverter Starting state

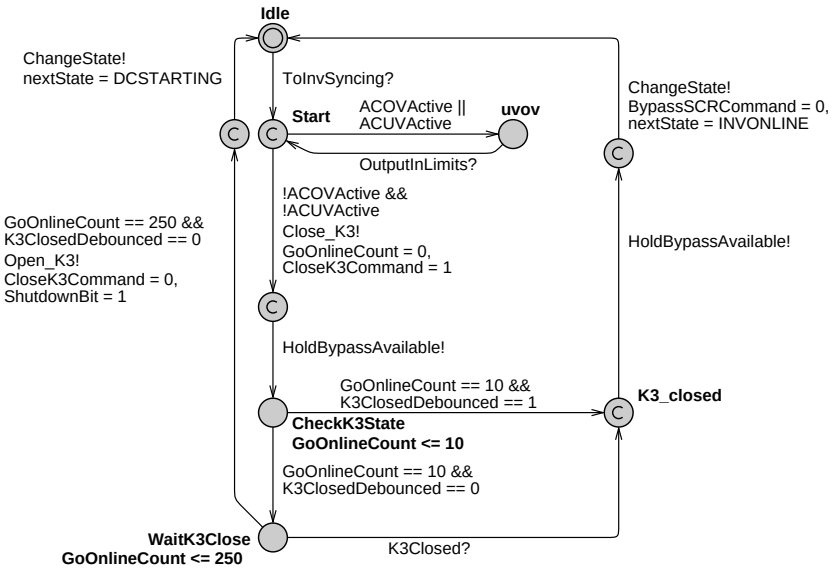

Figure 5.9: UPPAAL automaton modeling Inverter Syncing state

the Inverter Starting state.

Inverter Starting State

The timed automaton model of the Inverter Starting state is presented in Figure 5.8. A clock variable `c` is declared locally. After Inverter Starting state is entered, the inverter is turned on by assigning the variable `InvOn` to value 1. After that 5 seconds is waited in the bottommost location before moving back to location `Idle` and sending a synchronization `ChangeState` to the main state machine automaton to move to the next state.

Inverter Syncing State

A timed automaton modeling the Inverter Syncing state is shown in Figure 5.9. When the Inverter Syncing state is reached, the automaton transfer to location `Start`. If there is over- or undervoltage in the UPS output, the automaton transfers to a location `uvov` where it waits for the output voltage to get back within certain limits. When the output voltage is in limits, a synchronization `OutputInLimits` is received from the environment automaton and an edge back to location `Start` is fired. From there the automaton immedi-

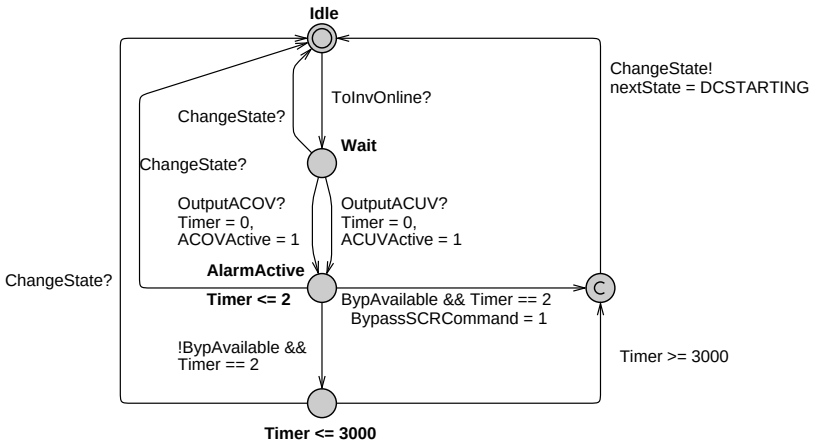

Figure 5.10: UPPAAL automaton modeling Inverter Online state

ately transfers forward sending a synchronization to close the K3 switch. After that a `HoldBypassAvailable` synchronization is sent to Bypass automaton, which holds the bypass available for one second after the UPS has transfered from bypass to inverter.

In the next location, named `CheckK3State`, the automaton waits for 10 ms for the K3 switch to close. If after those 10 ms K3 is closed, an edge to a committed location `K3_closed` is taken. As K3 is closed, the UPS is ready to transfer to the Inverter Online state. From the Inverter Syncing state it is possible to transfer also to the DC Starting state, which is why the variable `nextState` needs to be assigned to a value `INVONLINE` when sending the synchronization `ChangeState` to the main state machine automaton.

The other outgoing edge from the `CheckK3State` location leads to a location `WaitK3Close` and is taken if the K3 switch has not had time to close within 10 ms. If K3 closes (`K3Closed` synchronization is received), an edge to the location `K3_closed` is taken. Finally, if K3 is not closed after 250 ms from the close command, the automaton transfers back to location `Idle` via a committed location. In the first edge K3 is commanded open and a variable `ShutdownBit` is assigned to value 1 to signal the failure of K3. The main state machine is transferred to state DC Starting and, therefore, the `nextState` variable is assigned to value `DCSTARTING` in the edge sending the `ChangeState` synchronization.

Inverter Online State

The Inverter Online state is modeled with two automata, each handling its own set of actions and alarms. As the automata operate in parallel and both automata can send the `ChangeState` synchronization when transferring to the next state, they must be able to react to the `ChangeState` synchronization sent by the other automaton. This means that from each location, where time is allowed to pass, there must be an edge back to location `Idle` with a receiving synchronization from channel `ChangeState`. Those edges make sure that both automata are in location `Idle` when the main state machine is not in the Inverter Online state.

In Figure 5.10 is shown an automaton handling the inverter output under-

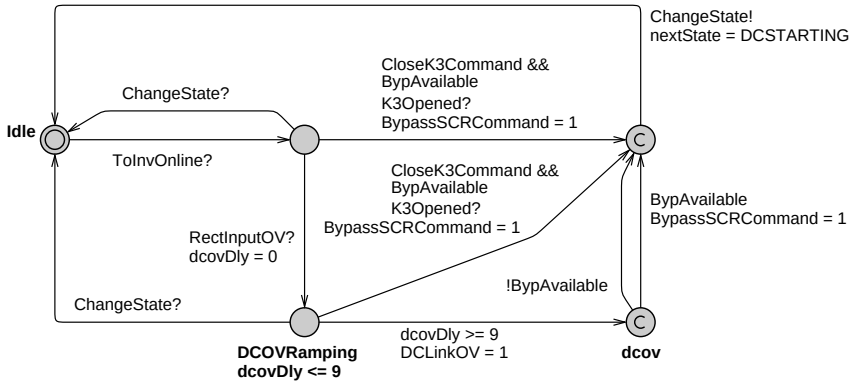

Figure 5.11: UPPAAL automaton modeling emergency transfer to bypass

and overvoltage alarms. The main operational criterion of the UPS in case of output under- or overvoltage is to transfer to bypass if it is available or in an extreme case of bypass not being available to drop the load.

If one of the voltage alarms becomes active (synchronization `OutputACOV` or `OutputACUV` is received from the Environment automaton) the automaton waits for 2 ms in location `AlarmActive` and checks whether bypass is available or not. If bypass is available, an edge leading to the committed location on the right is taken and the bypass static switch is commanded closed (`BypassSCRCommand = 1`). If bypass is not available, from the location `AlarmActive` an edge to the bottommost location is fired. In that location the automaton waits until 3 seconds has passed from the under-/overvoltage alarm becoming active and after that transfers to the committed location on the right. From the committed location an edge back to the location `Idle` is taken and the main state machine is commanded to transfer to the DC Starting state.

The other automaton, shown in Figure 5.11, models the overvoltage of the DC link and the case of K3 switch opening when it should be closed. If there is overvoltage in the input of the rectifier, `RectInputOV` synchronization is received from Environment automaton and an edge to location `DCOVRamping` is taken. The DC link voltage starts ramping up and after 9 milliseconds it goes over an overvoltage limit and the automaton takes an edge to location `dcov` assigning variable `DCLinkOV` to value 1. The UPS transfer to bypass, i.e., closes the bypass static switch, only if the bypass is available (modeled with the two outgoing edges from the location `dcov`). After that the automaton takes an edge back to location `Idle` telling the main state machine to transfer to the DC Starting state.

If the K3 switch opens when it is commanded closed (`K3Opened` synchronization is received although the variable `CloseK3Command` has a value of 1) and the bypass is available, the bypass static switch is closed. Also in the case of K3 opening spuriously the main state machine transfers to the DC Starting state.

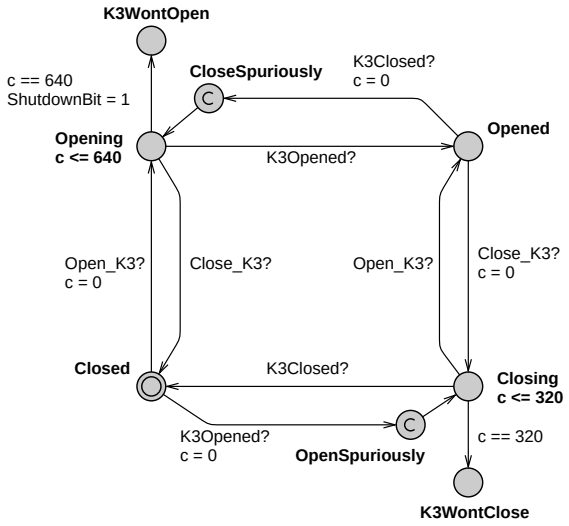

Figure 5.12: UPPAAL automaton observing the state of the K3 switch

5.5.2 Modeling the Control Software of Switches K3 and K5

K3 Switch

The software model of the K3 switch consists of two automata. An automaton observing the state of the switch is presented in Figure 5.12. The actual control logic of the switch is distributed around the model to other automata because in the source code the K3 switch is controlled from many places. Thus, implementing the control logic within one automaton would be difficult.

The automaton in Figure 5.12 has locations `Opened` and `Closed` modeling the state of the switch to be open or closed. Two intermediate locations, `Opening` and `Closing`, model the situations of the switch being commanded to open or close but having not had time to realize the command. The automaton receives synchronization events from several channels and transfers between the four aforementioned locations according to the synchronizations received. When the switch is commanded to open or close, a synchronization is received from channel `Open_K3` or `Close_K3`, respectively. When the switch opens or closes, a synchronization sent by the automaton modeling the hardware of K3 is received from channel `K3Opened` or `K3Closed`, respectively. Two committed locations, `OpenSpuriously` and `CloseSpuriously`, model the situation of the switch opening or closing when not commanded. The locations `K3WontOpen` and `K3WontClose` model the situations of the switch not opening or closing when commanded.

Another automaton, presented in Figure 5.13, updates the value of a `K3ClosedDebounced` variable. The variable has some filtering to make sure the state of the switch has really changed before updating the value of the variable. The automaton transfers between locations `Open` and `Closed` whenever it receives a synchronization from channel `K3Opened` or `K3Closed`. During those transitions the clock variable `c` is reset and after the automaton has been in one of those locations for the filtration time, an edge leading back to the location is taken updating the value of the `K3ClosedDebounced` vari-

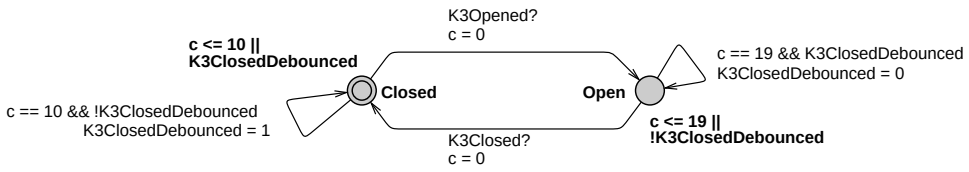

Figure 5.13: UPPAAL automaton updating `K3ClosedDebounced` variable

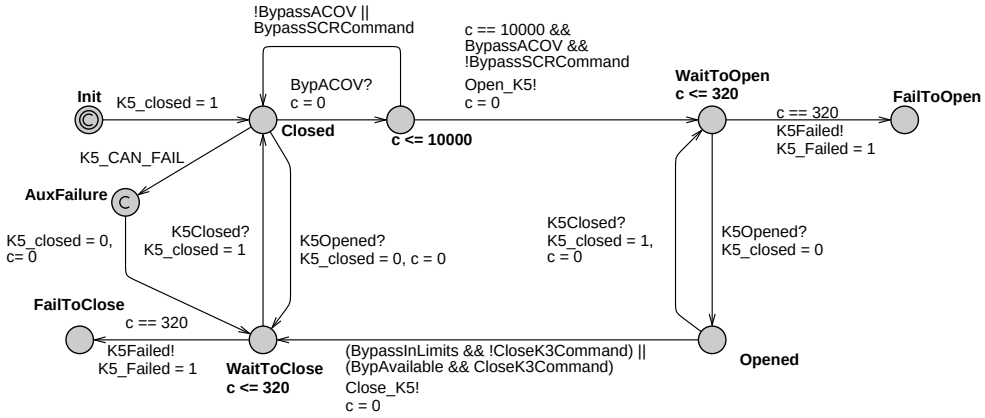

Figure 5.14: UPPAAL automaton modeling the control logic of K5 switch

able. The switch needs to be open for 10 ms and closed for 19 ms before the variable is updated.

K5 Switch

The software model of the K5 switch is shown in Figure 5.14. The automaton sends open and close commands to the hardware model of K5 and keeps track of the state of the switch.

Initially, the switch is assumed to be closed and from the initial location the automaton takes without delay an edge to location `Closed` assigning the variable `K5_closed` to value 1. If K5 is closed and the bypass overvoltage alarm becomes active (`BypACOV` synchronization is received from the Environment automaton), the automaton waits for 10 seconds and makes sure that the overvoltage alarm is still active and the bypass static switch is commanded open before giving a command to open the switch (`Open_K5` synchronization is sent to the hardware model of K5) by taking the edge to the location `WaitToOpen`. In the `WaitToOpen` location the automaton waits for the switch to open and when it happens (`K5Opened` synchronization is received from the hardware model automaton of K5), an edge to location `Opened` is taken. However, if the switch does not open within 320 ms, an edge to location `FailToOpen` is taken and a `K5Failed` synchronization is sent.

When the switch is open, it is closed again when certain conditions hold. If the voltage in the bypass line is within certain safe limits and the K3 switch is commanded to open or if the bypass is available and K3 is commanded closed, a command to close K5 is given and the automaton transfers to location `WaitToClose`. The channel `Close_K5` is declared as *urgent* so that the

edge from Opened to WaitToClose is taken as soon as the guard of the edge becomes true. As in the case of opening the switch, at most 320 ms is waited in the location WaitToClose for the switch to close and an edge either to location FailToClose or to location Closed is taken.

Another fault mode of the switch (in addition to not opening or closing within certain time limits) is to open or close when not commanded to. This is modeled with the edge from the location Closed to WaitToClose for the case of K5 opening spuriously and with the edge from the location Opened to WaitToOpen for the case of K5 closing spuriously. Finally, the auxiliary switch (showing if the switch is closed or not) of K5 can fail showing that the switch is open although it is closed. This failure is modeled with the location AuxFailure, which can be reached from the location Closed any time. From AuxFailure an edge is immediately taken to location WaitToClose and the variable K5_closed is assigned to value 0 to model the situation of software seeing the switch to be closed.

5.5.3 Modeling the Bypass

The operation of bypass is modeled with two automata. The main automaton keeping track of the availability of the bypass is shown in Figure 5.15. Initially, the bypass is assumed to be available and the bypass is connected to the output (K5 and bypass static switch are closed). Thus, from the initial location an edge is taken to location Available assigning variables BypAvailable and BypassSCRCommand to one. If the K5 switch fails to open or close (K5Failed synchronization is received) or if there is overvoltage in the bypass line (BypACOV synchronization is received), the bypass is made unavailable. The delay before bypass becomes unavailable is between 10 and 15 ms (waited in location Filtering) because the availability of the bypass is handled in a function called every 5 ms and there is a short filtering before the UPS takes action in those fault situations. The varying delay is modeled as explained in the timing example in Section 4.3.2.

The outgoing edge from the location Unavailable is possible to take only if the value of the variable K5_Failed is 0, i.e., K5 has not failed to open or close and thus it is possible for the bypass to become available again. There is a delay of 0–5 ms before the bypass becomes available after the voltage in the bypass line has dropped back to nominal.

After transferring from bypass to inverter the bypass is kept available for one second. This is modeled with the edge from the location Available to HoldAvailable with a synchronization HoldBypassAvailable sent by the automaton modeling the Inverter Syncing state during the transfer from bypass to inverter. In HoldAvailable the automaton stays for one second and after that transfers back to the location Available. However, if during the period of one second the voltage in the bypass line has raised over the overvoltage threshold or K5 has failed, the bypass is made unavailable by transferring to location Unavailable via the location Filtering.

An automaton modeling the overvoltage of the bypass is shown in Figure 5.16. If the ENABLE_BYP_ACOV configuration bit is set to 0, the automaton cannot leave the location InLimits and thus the configuration bit prevents bypass overvoltage from happening. When the overvoltage is en-

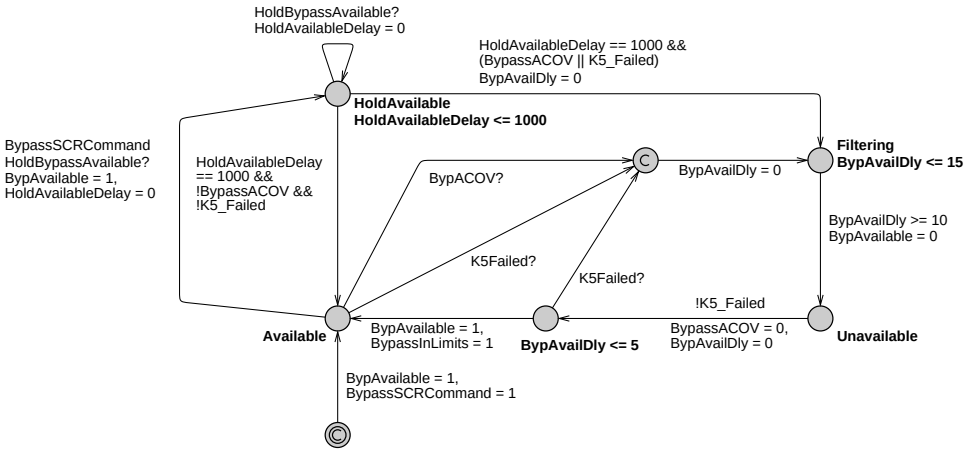

Figure 5.15: UPPAAL automaton modeling the bypass

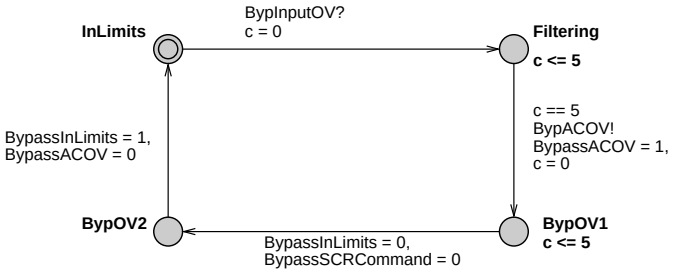

Figure 5.16: UPPAAL automaton modeling the bypass overvoltage

abled by the configuration bit, a transition from the location `InLimits` to location `Filtering` is taken when the environment automaton sends the `BypInputOV` synchronization. It takes about five milliseconds before the bypass overvoltage alarm is set because of a couple of filtrations in the UPS firmware. This is why the automaton waits in the location `Filtering` for five time units before transferring to location `BypOV1`. Within five milliseconds from that the next edge to location `BypOV2` is taken. During the transition the bypass static switch is opened (`BypassSCRCommand = 0`). From `BypOV2` the edge back to location `InLimits` can be taken anytime. The transition models the situation of bypass voltage coming back to nominal level.

5.6 VERIFIED PROPERTIES

The UPPAAL model of the control firmware of the UPS device was verified against several properties derived from a set of failure cases. In this section the properties are introduced and in the following chapter the verification results are discussed.

5.6.1 Failure Case 1

Failure description: K3 will not close when commanded.

Action: UPS should stay on bypass if the bypass is available. In UPPAAL's TCTL the property is formalized as follows:

```
A[] ((k3_sw.K3WontClose && BypAvailable &&
State.InvSyncingState) imply (K5_closed &&
BypassSCRCommand))
```

The formula states that always when the K3 switch is not closed although commanded (`k3.K3WontClose`), the bypass is available (`BypAvailable`), and the main state machine is in Inverter Syncing state, then the load should be on bypass. The load is on bypass when the bypass backfeed contactor is closed (`K5_closed`) and the static bypass switch is commanded closed (`BypassSCRCommand`). The main state machine is required to be in the Inverter Syncing state because that is the only state where the UPS attempts to close the K3 switch and the location `k3.K3WontClose` is also reachable in the case of K3 opening when it should be closed.

5.6.2 Failure Case 2

Failure description: K3 opens when it should be closed.

Action: The desired action depends on whether the bypass is available or not.

a. If the load is on inverter and the bypass is available, the UPS should transfer to bypass within 20 ms. In TCTL this is:

```
State.InvOnLineState && CloseK3Command &&
k3_sw.OpenSpuriously && BypAvailable -->
k3_sw.c <= 20 && K5_closed && BypassSCRCommand
```

The formula states that when K3 is commanded closed (`CloseK3Command`) and it opens when it should not (`k3.OpenSpuriously`) and the bypass is available, then eventually K5 should be closed and the static switch should be commanded closed. The clock `k3.K3OpenedCounter` is reset when K3 opens, and before the clock reaches value 20 (ms), the bypass switches should be closed.

b. If the load is on inverter and the bypass is not available, there should be no changes in the operation of the UPS. In TCTL this is formalized as follows:

```
A[] (CloseK3Command && (k3_sw.OpenSpuriously ||
k3_sw.Opened) && !BypAvailable imply !BypassSCRCommand)
```

The formula states that always when K3 is commanded closed but it is open and the bypass is not available, the bypass static switch is commanded open.

5.6.3 Failure Case 3

Failure description: K3 will not open when commanded or closes when it should be open.

Action: If the load is on bypass, the UPS should stay on bypass without testing the inverter. In TCTL:

```
A[] !(k3.K3WontOpen && InvOn)
```

The formula states that the system should never reach a state where K3 will not open (`k3.K3WontOpen`) and the inverter is commanded on (`InvOn`).

5.6.4 Failure Case 4

Failure description: K5 wont close or opens when should be closed.

Actions: If the load is on inverter, stay on inverter and disable bypass operation. The property is checked in two parts.

a. First we check that the UPS stays on inverter. In TCTL:

```
A[] (k5.FailToClose && !ACOVActive && !ACUVActive &&
!DCLinkOV && State.InvOnLineState) imply
(InvOn && K3ClosedDebounced)
```

The formula contains several extra constraints to exclude inverter and DC link failures from happening at the same time. The formula specifies that whenever k5 automaton is in `FailToClose` location (the location is also reached when K5 opens when it should not), output AC under- and overvoltage alarms are not active, there is no overvoltage in the DC link (`!DCLinkOV`), and the main state machine is in `InvOnLineState`, then the inverter should be on and K3 closed (`K3ClosedDebounced`).

b. The second formula verifies that the bypass operation is disabled in case of K5 failure. In TCTL:

```
k5.FailToClose --> !BypAvailable
```

The formula states that whenever the K5 fails to close or K5 opens when it should be closed, eventually the bypass operation is disabled (`!BypAvailable`).

5.6.5 Failure Case 5

Failure description: The auxiliary switch of K5 changes to open state when the contactor is closed and commanded to be closed.

Action: The UPS should continue feeding the load on bypass. In TCTL this is:

```
A[] (k5_hw.Closed && !K5_closed && !CloseK3Command &&
BypAvailable imply BypassSCRCommand)
```

If the auxiliary switch of K5 shows that the switch is open (`!K5_closed`) although it is closed (`k5_hw.Closed`), the UPS is not attempting to feed the load with inverter (K3 is commanded open: `!CloseK3Command`), and the bypass is available, then the UPS stays on bypass keeping the bypass static switch closed.

5.6.6 Failure Cases 6-9

Failure description: Inverter output over/undervoltage. Caused by either:

- **Failure case 6:** Inverter output capacitor shorts in one phase -> Under voltage
- **Failure case 7:** Inverter output voltage feedback fails in one phase, the measurement halts to zero voltage value
- **Failure case 8:** Inverter IGBT shorts, output is shorted to DC link -> Over voltage
- **Failure case 9:** Inverter output voltage feedback fails in one phase, the measurement halts to the maximum voltage value

Action: UPS reacts to each of the four over- and undervoltage cases the same way and, thus, they can be checked using the same properties. The desired behavior on failure depends on whether the load is on inverter or bypass and the availability of bypass.

a. If the load is on inverter and the bypass is available, the UPS should eventually transfer to bypass. In failure cases 7 and 8 the transfer should happen within 5 ms from the fault, and in cases 6 and 9 within 10 ms from the fault. However, in each of these cases 5 ms is used in the verification as it proved to be sufficient also for cases 6 and 9. With UPPAAL's TCTL the property is specified as:

```
(acuvov.AlarmActive && BypAvailable) --->
(acuvov.Timer <= 5 && BypassSCRCommand &&
(K5_closed || k5.Closing))
```

The property states that if output under- or overvoltage alarm is active and bypass is available, before 5 ms from the alarm (`acuvov.Timer <= 5`) the static switch is closed and K5 is either closed or closing. The clock variable `acuvov.Timer` is reset immediately after the alarm becomes active. K5 can not be assumed to be closed after 5 ms because it takes between 30 and 115 ms for the switch to close.

b. The second case is that the load is on inverter but the bypass is not available, in which case the output must be turned off within 3 seconds. In TCTL this is:

```
(acuvov.AlarmActive && !BypAvailable) --->
(!BypassSCRCommand && !K5_closed &&
!K3ClosedDebounced && acuvov.Timer <= 3000)
```

c. If the load is on bypass when the failure occurs, the UPS should test the inverter by turning it on and stay on bypass if the inverter output is not within limits. In other words the UPS should transfer back to inverter when there is no reason to stay on bypass. In TCTL the property is:

```
(BypassSCRCommand && !ACOVActive && !ACUVActive &&
!DCLinkOV) --> (CloseK3Command && InvOn)
```

The UPS should transfer from bypass to inverter when output under- and overvoltage and DC link overvoltage alarms are not active.

5.6.7 Failure Case 10

Failure description: Symmetric input overvoltage. If the bypass input line and the rectifier are connected to a common supply source, both DC link and bypass voltages rise simultaneously.

Action: The desired action depends on whether the bypass input is connected to the same supply source as the rectifier and whether the bypass is available or not.

a. In case of symmetric input overvoltage UPS should not transfer to bypass. In TCTL:

```
A[] !(DCLinkOV && BypassACOV && BypassSCRCommand &&
K5_closed)
```

The TCTL property states that there should never be a situation when DC link overvoltage and bypass overvoltage alarms are active and the UPS is on bypass (static switch and K5 closed).

b. If the bypass supply is not from the same source with the rectifier (there is no overvoltage in the bypass line), transfer to bypass. In TCTL:

```
(etb.dcov && BypAvailable && !BypassACOV) -->
(BypassSCRCommand && K5_closed)
```

The formula states that if there is overvoltage in the DC link but the bypass is available and no overvoltage is present in bypass line, then eventually the UPS is on bypass, i.e., the bypass static switch and the K5 switch are both closed.

Table 6.1: Model checking times

Failure Case	Configuration			Time	Satisfied
	K3	K5	Bypass		
1	1	0	0	<1 s	Yes
2 a	1	0	0	<1 s	Yes
2 b	1	0	0	<1 s	Yes
3	1	0	0	<1 s	No
4 a	0	1	0	5 s	Yes
4 b	0	1	0	34 s	Yes
5	0	1	0	5 s	Yes
6–9 a	0	0	0	<1 s	Yes
6–9 b	0	0	0	<1 s	Yes
6–9 c	0	0	0	<1 s	Yes
10 a	0	0	1	<1 s	No
10 b	0	0	0	<1 s	Yes

6 RESULTS

The model checking was performed on a standard PC with 2 GB of RAM and Intel Core 2 Duo E6320 processor running at 1.86 GHz. The UPPAAL version used in the model checking was 4.0.11 and the operating system was Debian GNU/Linux 4.0.

The verification proved that in most of the failure cases the UPS firmware behaves correctly. The verification times were within one minute for all of the verified properties as seen from Table 6.1. The verification times did not vary a lot even though different configurations were used for checking different properties. In Table 6.1 the configurations are presented by the values of the three configuration variables: K3, K5, and Bypass represent the variables `K3_CAN_FAIL`, `K5_CAN_FAIL`, and `ENABLE_BYP_ACOV`, respectively.

As comparison, in Table 6.2 model checking times are shown for a model in which all the failures are enabled (by using the following configuration: `K3_CAN_FAIL = 1, K5_CAN_FAIL = 1, ENABLE_BYP_ACOV = 1`). From the results it can be seen that most of the properties are not satisfied any longer. This is caused by the additional failure modes introducing functionality causing the properties to be violated.

From the results in Table 6.2 it can also be seen that some of the properties have much longer verification times than when checked against a single-fault model. This is caused by the fact that the additional failure modes make the state space of the model substantially larger. Especially, the verification time of the property of failure case 4b grew enormously; the verification did not finish within 20 hours. This is caused by the added functionality and several time delays of different scale in the model causing the executions of the model to have a great number of states.

The size of the state space varies depending on the verified property. In the model checking runs performed for the safety properties (properties of form `A[]` ϕ) of failure cases 1–10 the size of the state space varied between

Table 6.2: Model checking times when all failures were enabled

Failure Case	Time	Satisfied
1	1 min 29 s	No
2 a	2 s	No
2 b	1 s	No
3	<1 s	No
4 a	1 s	No
4 b	>20 h	–
5	19 s	No
6–9 a	1 s	No
6–9 b	2 s	No
6–9 c	12 min 13 s	No
10 a	1 s	No
10 b	11 s	No

50 and 205,000. The state space of liveness properties (properties of form ϕ `-->` ψ) is not known because UPPAAL does not support the calculation of the state space size for those properties.

As a result of the model checking failure cases 3 and 10a were found to be violated. In failure case 3 the property does not hold because the inverter can be turned on although K3 is closed. In the Inverter Starting state the inverter is turned on without checking if the K3 switch is open. The firmware has a functionality to prevent the starting of the inverter, but it is based on a filtered bit of K3 being closed although commanded open with a filtering time of 640 ms. During that time the state machine can reach the Inverter Starting state where the inverter is turned on. This happens if the K3 switch closes when the state machine is transferring from the DC Starting state to the Inverter Starting state and the transfer happens within 640 ms from the switch closing.

The failure case 10a is violated because the UPS may transfer to bypass even though there is overvoltage in the bypass line. The situation of symmetric input overvoltage is shown in Figure 6.1.

The decision to transfer to bypass is made after the `DCLinkOV` variable gets value 1. The decision is made based on the value of the variable `BypAvailable`, as modeled in the automaton modeling the DC link overvoltage (shown in Figure 5.11). The edge assigning the variable `BypassSCRCommand` to 1 is taken only if the bypass is available.

The values of the variables regarding the bypass operation in the case of bypass overvoltage are assigned as follows. The variable `BypACOV` is assigned to 1 after 5 ms from the beginning of the overvoltage situation as modeled by the bypass overvoltage automaton shown in Figure 5.16. From this it follows that the variable `BypassInLimits` is assigned to 0 value after 0–5 ms (see the automaton in Figure 5.16) and the `BypAvailable` variable is assigned to 0 after 10–15 ms (in the automaton in Figure 5.15).

The bypass static switch is opened only after the `BypassInLimits` variable has changed to value 0. This means that there exists a timing sequence where the load is briefly transfered to bypass although there is overvoltage in the

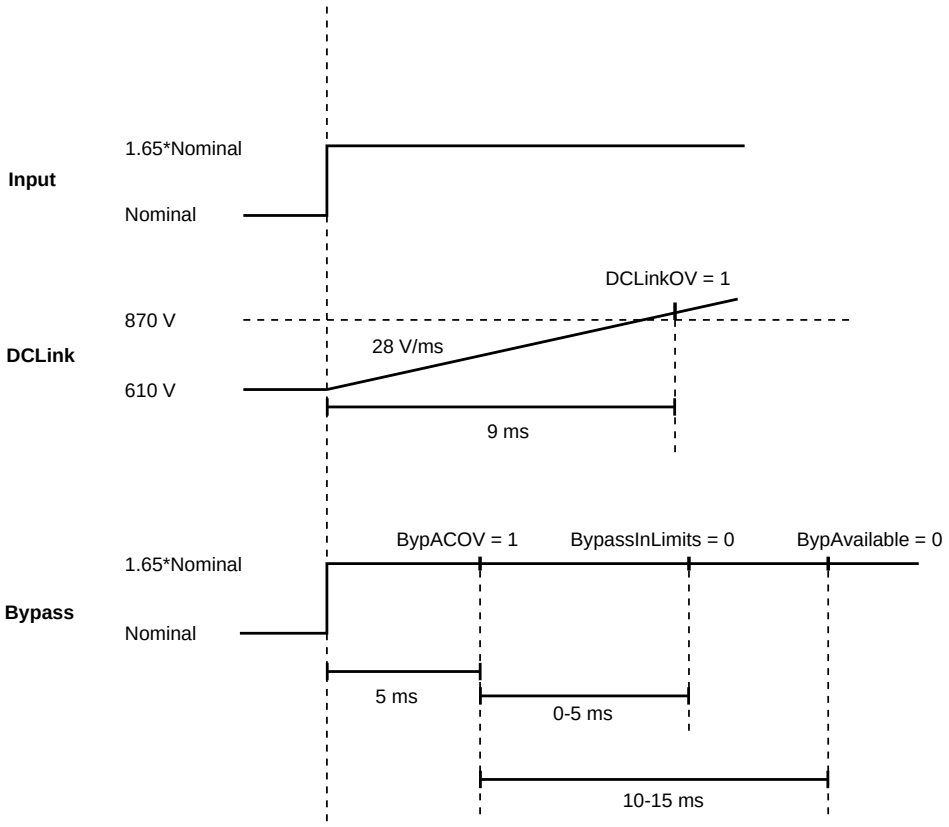

Figure 6.1: Voltages and bit assignments in case of symmetric input overvoltage

bypass line. The decision to transfer to bypass is made before the UPS has had time to react to the overvoltage in the bypass line.

7 CONCLUSIONS

In this work we studied the applicability of model checking to the verification of real-time embedded control software. The central goal of the study was to evaluate whether a real-world embedded software can be modeled with adequate level of abstraction so that the verification results are reliable. The suitability of the real-time model checking tool UPPAAL for the verification of the control software of a UPS device was evaluated.

The modeling of an embedded software is not a straightforward task. Finding all different control sequences from the code can be a time consuming and error prone task. Modeling a variable which is updated in dozens of places and read in even more is challenging. It is also difficult to verify that the model has all the same behaviors as the system itself. This is why systematic model checking methodologies are needed for modeling an embedded software. In this work we presented a methodology for modeling and analyzing a real-time embedded software. The methodology makes it easier to create modular and easily reviewable models.

In the case study a part of the control firmware of a UPS device was modeled using the UPPAAL model checker. Several failure cases were formalized using UPPAAL's TCTL and verified against the model. The firmware was found to operate as desired in most of the failure cases. However, in two of the failure cases a timing-related bug was revealed with model checking.

UPPAAL was found to be a suitable tool for model checking a timed embedded control software. The model of the UPS device consists of a total of 16 clock variables and the time delays range from a couple of milliseconds to ten seconds making the timing of the model rather complex. Even then, UPPAAL was able to carry out the model checking of each of the verified properties within one minute in the case of single-failure models. In conclusion, model checking is a valuable tool for finding bugs from embedded software and, on the other hand, for verifying whether the design of a complex embedded system meets its requirements.

7.1 FUTURE WORK

The next step is to develop more automated methods for translating the source code to the UPPAAL model. Automated translation is needed for utilizing model checking as part of the software development process to find bugs and design errors already during the development phase. Building a model manually is too time consuming if the software is modified often. By using automated tools for the translation a new revision of the code can be easily translated to a model and checked against properties, which verify that there are no bugs or regressions in the code.

The applicability of the model checking methodology presented in this work should be assessed against other case studies. The structure of an embedded software varies largely from application to another, which makes it difficult to establish modeling techniques applicable to a wide range of systems.

ACKNOWLEDGEMENTS

This report is a reprint of my Master's Thesis. This work was prepared under the research project Model-Based Safety Evaluation of Automation Systems (MODSAFE), which is part of the Finnish Research Programme on Nuclear Power Plant Safety 2007–2010 (SAFIR2010). I am very grateful to my supervisor Prof. Ilkka Niemelä for giving me the opportunity to work in this research project and for the valuable guidance and encouragement during the preparation of this Thesis. I also want to thank Prof. Keijo Heljanko for his comments and advice.

Special thanks go to Eaton Power Quality Oy for providing material used in the case study and to Jari Eloranta and Risto Karola for support and advice during the study.

Finally, I want to thank my family for their support throughout the period of my university studies.

BIBLIOGRAPHY

[1] Aditya Agrawal, Gabor Karsai, and Akos Ledeczi. An end-to-end domain-driven software development framework. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 2003)*, pages 8–15. ACM, 2003.

[2] Rajeev Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford University, 1991.

[3] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[4] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, pages 322–335. Springer-Verlag, 1990.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[6] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM 2004)*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2004.

[7] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the ACM SIGPLAN Symposium on the Principles of Programming Languages (POPL 2002)*, pages 1–3. ACM, 2002.

[8] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems (revised lectures)*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer-Verlag, 2004.

[9] Stoyan B. Bekiarov and Ali Emadi. Uninterruptible power supplies: classification, operation, dynamics, and control. In *Proceedings of the 17th Annual IEEE Applied Power Electronics Conference (APEC 2002)*, pages 597–604, 2002.

[10] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124. Springer-Verlag, 2004.

[11] Patricia Bouyer. Weighted timed automata: Model-checking and games. In *Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXII)*, volume

158 of *Electronic Notes in Theoretical Computer Science*, pages 3–17, 2006.

[12] Bart Broekman and Edwin Notenboom. *Testing Embedded Software*. Pearson Education Limited, 2003.

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[14] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proceedings of the 11th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer-Verlag, 2005.

[15] Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous Boolean programs. In P. Godefroid, editor, *Proceedings of SPIN 2005*, volume 3639 of *Lecture Notes in Computer Science*, pages 75–90. Springer-Verlag, 2005.

[16] L. de Moura, S. Owre, and N. Shankar. The SAL language manual. Technical Report SRI-CSL-01-02 (Rev. 2), SRI International, 2003.

[17] A. Emadi, A. Nasiri, and Stoyan B. Bekiarov. *Uninterruptible Power Supplies and Active Filters*. CRC Press, 2005.

[18] E. A. Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[19] Colin Fidge and Phil Cook. Model checking interrupt-dependent software. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pages 51–58. IEEE Computer Society, 2005.

[20] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1997.

[21] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, volume 600 of *Lecture Notes in Computer Science*, pages 226–251, 1991.

[22] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Journal of Information and Computation*, 111(2):193–244, 1994.

[23] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[24] M. Jurdzinski, F. Laroussinie, and J. Sproston. Model checking probabilistic timed automata with one or two clocks. In *Proceedings of the 13th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, volume 4424 of *Lecture Notes in Computer Science*, pages 170–184. Springer-Verlag, 2007.

[25] Pim Kars. Formal methods in the design of a storm surge barrier control system. *Lecture Notes in Computer Science: Lectures on Embedded Systems*, 1494:353–367, 1998.

[26] G. Karsai, S. Neema, A. Bakay, A. Ledeczi, F. Shi, and A. Gokhale. A model-based front-end to TAO/ACE: The embedded system modeling language. In *Proceedings of the 2nd Workshop on TAO*, 2002.

[27] Jean J. Labrosse, Jack G. Ganssle, Robert Oshana, Colin Walls, and Keith E. Curtis. *Embedded Software*. Newnes, 2007.

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

[29] Gabor Madl, Sherif Abdelwahed, and Gabor Karsai. Automatic verification of component-based real-time CORBA applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS 2004)*, pages 231–240. IEEE Computer Society, 2004.

[30] Gabor Madl, Sherif Abdelwahed, and Douglas C. Schmidt. Verifying distributed real-time properties of embedded systems via graph transformations and model checking. *Real-Time Systems*, 33(1-3):77–100, 2006.

[31] K. McMillan. *Getting started with SMV*. Cadence Berkeley Laboratories, USA, 1998.

[32] Madhavan Mukund. Finite-state automata on infinite inputs. In *The 6th National Seminar on Theorectical Computer Science*, Banasthali, Rajasthan, India, 1996.

[33] Bastian Schlich and Stefan Kowalewski. [mc]square: A model checker for microcontroller code. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2006)*, pages 466–473. IEEE Computer Society, 2006.

[34] Bastian Schlich, Michael Rohrbach, Michael Weber, and Stefan Kowalewski. Model checking software for microcontrollers. Technical Report AIB-2006-11, RWTH Aachen University, 2006.

[35] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN 2005)*, pages 477–482. IEEE Press, 2005.

[36] Jan Tretmans, Klaas Wijbrans, and Michel Chaudron. Software engineering with formal methods: The development of a storm surge barrier control system - revisiting seven myths of formal methods. *Formal Methods in System Design*, 19(2):195–215, 2001.

[37] Antti Valmari. The state explosion problem. *Lecture Notes in Computer Science: Lectures on Petri Nets I: Basic Models*, 1491:429–528, 1998.

[38] M. Weber. An embeddable virtual machine for state space generation. In D. Bošnački and S. Edelkamp, editors, *Proceedings of the 14th SPIN Workshop*, volume 4595 of *Lecture Notes in Computer Science*, pages 168–185. Springer-Verlag, 2007.

[39] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent Linux device drivers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE 2007)*, pages 501–504. ACM, 2007.

[40] Sergio Yovine. Kronos: A verification tool for real-time systems. (Kronos user's manual release 2.2). *International Journal on Software Tools for Technology Transfer*, 1:123–133, 1997.

A GLOBAL VARIABLE DECLARATIONS OF THE UPPAAL MODEL OF THE UPS DEVICE

```
bool K3ClosedDebounced;
bool K5_closed;
bool CloseK5Command;
bool BypAvailable;
bool BypassACOV;
bool BypassInLimits;
bool CloseK3Command;
bool BypassSCRCommand;
bool DCLinkOV;
bool InvOn;
bool ACOVActive;
bool ACUVActive;
bool ShutdownBit;
bool K5_Failed;

broadcast chan Open_K3;
broadcast chan Close_K3;
broadcast chan K3Opened;
broadcast chan K3Closed;
chan Open_K5;
urgent chan Close_K5;
broadcast chan K5Opened;
broadcast chan K5Closed;
broadcast chan K5Failed;

chan OutputACOV;
chan OutputACUV;
broadcast chan BypACOV;
chan RectInputOV;
broadcast chan BypInputOV;
broadcast chan HoldBypassAvailable;
chan DCInLimits;
chan OutputInLimits;

broadcast chan ChangeState;
chan ToDCStarting;
chan ToInvStarting;
chan ToInvSyncing;
broadcast chan ToInvOnline;

// Variables for modeling the state machine
int nextState;
const int DCSTARTING = 0;
const int INVSYNCING = 1;
const int INVONLINE = 2;

// Configuration bits
const bool K3_CAN_FAIL = 0;
```

```
const bool K5_CAN_FAIL = 1;
const bool ENABLE_BYP_ACOV = 0;
```

B SYSTEM DECLARATIONS OF THE UPPAAL MODEL OF THE UPS DEVICE

```
// Process instantiations of the environment model
// automata
env = Environment();
// Parameters: min and max closing time,
// min and max opening time
k3_hw = K3_HW(30, 115, 25, 80);
k5_hw = K5_HW(30, 115, 25, 80);

// Process instantiations of the software model
// automata
State = MainStateMachine();
dcstarting = DCStartingState();
invstarting = InvStartingState();
invsyncing = InvSyncingState();
invonline = InvOnlineState();

k3_sw = K3_SW();
k3_cd = K3CD();
k5_sw = K5_SW();
bypass = Bypass();
byp_ov = BypassOV();

etb = EmergXferToByp();

// List of processes to be composed into a system.
system State, dcstarting, invstarting, invsyncing,
   invonline, k3_sw, k3_hw, k3_cd, k5_sw, k5_hw, env,
   bypass, byp_ov, etb;
```

TKK REPORTS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-R18 Roland Kindermann
Testing a Java Card applet using the LIME Interface Test Bench: A case study.
September 2009.

TKK-ICS-R19 Kalle J. Palomäki, Ulpu Remes, Mikko Kurimo (Eds.)
Studies on Noise Robust Automatic Speech Recognition. September 2009.

TKK-ICS-R20 Kristian Nybo, Juuso Parkkinen, Samuel Kaski
Graph Visualization With Latent Variable Models. September 2009.

TKK-ICS-R21 Sami Hanhijärvi, Kai Puolamäki, Gemma C. Garriga
Multiple Hypothesis Testing in Pattern Discovery. November 2009.

TKK-ICS-R22 Antti E. J. Hyvärinen, Tommi Junttila, Ilkka Niemelä
Partitioning Search Spaces of a Randomized Search. November 2009.

TKK-ICS-R23 Matti Pöllä, Timo Honkela, Teuvo Kohonen
Bibliography of Self-Organizing Map (SOM) Papers: 2002–2005 Addendum.
December 2009.

TKK-ICS-R24 Timo Honkela, Nina Janasik, Krista Lagus, Tiina Lindh-Knuutila, Mika Pantzar, Juha Raitio
Modeling communities of experts. December 2009.

TKK-ICS-R25 Jani Lampinen, Sami Liedes, Kari Kähkönen, Janne Kauttio, Keijo Heljanko
Interface Specification Methods for Software Components. December 2009.

TKK-ICS-R26 Kari Kähkönen
Automated Test Generation for Software Components. December 2009.

TKK-ICS-R27 Antti Ajanki, Mark Billinghurst, Melih Kandemir, Samuel Kaski, Markus Koskela, Mikko Kurimo, Jorma Laaksonen, Kai Puolamäki, Timo Tossavainen
Ubiquitous Contextual Information Access with Proactive Retrieval and Augmentation.
December 2009.

ISBN 978-952-60-3102-6 (Print)
ISBN 978-952-60-3103-3 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)