TKK Reports in Information and Computer Science

Espoo 2008 TKK-ICS-R3

# MODEL CHECKING TIMED SAFETY INSTRUMENTED SYSTEMS

Jussi Lahtinen

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

TKK Reports in Information and Computer Science
Espoo 2008 TKK-ICS-R3

# MODEL CHECKING TIMED SAFETY INSTRUMENTED SYSTEMS

**Jussi Lahtinen**

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
P.O.Box 5400
FI-02015 TKK
FINLAND
URL: http://ics.tkk.fi
Tel. +358 9 451 1
Fax +358 9 451 3369
E-mail: series@ics.tkk.fi



ISBN 978-951-22-9444-2 (Print)
ISBN 978-951-22-9445-9 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)
URL: http://www.otalib.fi/tkk/edoc/

TKK ICS
Espoo 2008

ABSTRACT: Defects in safety-critical software systems can cause large economical and other losses. Often these systems are far too complex to be tested extensively. In this work a formal verification technique called model checking is utilized. In the technique, a mathematical model is created that captures the essential behaviour of the system. The specifications of the system are stated in some formal language, usually temporal logic. The behaviour of the model can then be checked exhaustively against a given specification.

This report studies the Falcon arc protection system engineered by UTU Oy, which is controlled by a single programmable logic controller (PLC). Two separate models of the arc protection system are created. Both models consist of a network of timed automata. In the first model, the controller operates in discrete time steps at a specific rate. In the second model, the controller operates at varying frequency in continuous time. Five system specifications were formulated in timed computation tree logic (TCTL). Using the model checking tool Uppaal both models were verified against all five specifications.

The processing times of the verification are measured and presented. The discrete-time model has to be abstracted a lot before it can be verified in a reasonable time. The continuous-time model, however, covered more behaviour than the system to be modelled, and could still be verified in a moderate time period. In that sense, the continuous-time model is better than the discrete-time model.

The main contributions of this report are the model checking of a safety instrumented system controlled by a PLC, and the techniques used to describe various TCTL specifications in Uppaal. The conclusion of the work is that model checking of timed systems can be used in the verification of safety instrumented systems.

KEYWORDS: safety instrumented systems, model checking, real-time, Uppaal

## CONTENTS

## LIST OF FIGURES

## LIST OF SYMBOLS AND ABBREVIATIONS

| | |
|---|---|
| $\{d\}$ | The fractional part of $d$ |
| $\lfloor d \rfloor$ | The integer part of $d$ |
| $\mathbb{N}$ | The set of natural numbers |
| $\mathbb{R}_+$ | The set of non-negative real numbers |
| $\mathbb{R}^C$ | The set of clock valuations |
| A | Temporal logic path quantifier: for all computation paths |
| E | Temporal logic path quantifier: for some computation path |
| U | Temporal logic operator: until |
| X | Temporal logic operator: next time |
| BDD | Binary decision diagram |
| CCS | Calculus of Communicating Systems |
| CTL | Computation tree logic |
| DBM | Difference bound matrix |
| FBD | Function block diagram |
| IEC | International Electrotechnical Commission |
| IL | Instruction list |
| LD | Ladder diagram |
| PLC | Programmable logic controller |
| SFC | Structured function chart |
| SIS | Safety instrumented system |
| ST | Structured text |
| TCTL | Timed computation tree logic |
| TON | Timer on delay |
| UTU | Urho Tuominen Oy |

# 1 INTRODUCTION

Software plays an increasing role in safety-critical applications where an incorrect behaviour could lead to significant economical, environmental or personnel losses. Thus, it is imperative that these safety-critical systems conform to their functional requirements. Testing is regularly used to ensure that the requirements are met. Testing can not, however, show the absence of software bugs, only their presence. If the system functionality has to be verified, some much more powerful method is needed.

## 1.1 Model Checking

Model checking [20] is an automatic technique for verifying hardware and software designs. Other, more traditional system verification techniques include simulation, testing, and deductive reasoning [20]. Deductive reasoning normally means the use of axioms and proof rules to prove the correctness of systems. Deductive reasoning techniques are often difficult and require a lot of manual intervention. On the other hand, validation techniques based on extensive testing or simulation can easily miss errors when the number of possible states of the system is very large. Model checking requires no user supervision and always produces a counterexample when the design fails to satisfy some checked property.

Model checking consists of modeling, specification, and verification. Firstly, the design under investigation has to be converted into a formalism understood by the used model checking tool. This means that the system behaviour is depicted in a modeling language. The model should comprise the essential properties of the system, and at the same time abstract away from unimportant details that only complicate the verification process [20].

Secondly, the system has some properties it must satisfy. These properties, also called specifications, are usually given in some logical formalism. For hardware and software designs, it is typical to use temporal logic [20], which can express the requirements for system behaviour over time.

After modeling and specification, only the fully automatic model checker part remains. If the design meets the desired properties, the verification tool will state that the specification is true. In case of a design flaw or an incorrect modeling or specification, a counterexample will be generated. A counterexample presents a legal execution sequence in the model that is not allowed by a specification. The analysis of the counterexample is usually impossible to do automatically and thus involves human assistance. For example, it is impossible for a computer program to decide whether the model or the specification is incorrect. The counterexample can help the designer find the errors in the specifications, in the design or in the model.

There are several model checking techniques. Many of them suffer from the state explosion problem [46]. State explosion results from the fact that

the number of states in a system grows exponentially as the size of the model increases. Although the system is still finite, model checking might be too complex for even state-of-the-art computers. No fully satisfactory solution to this problem has yet been found, although symbolic representation of the state space using BDDs or reducing the needed state space using abstraction have been found useful [17, 46]. Partial order reduction [24, 46, 20] is also a typical state space reduction method. Bounded model checking [9, 10] attempts to avoid the state explosion problem by bounding the counterexample length. Nevertheless, model checking is likely to prove an invaluable tool to verify system requirements or design.

### 1.2 Work Description

In this work, a real-time safety-critical system is modelled as a network of timed automata [8]. Furthermore, the model is verified against five properties using the model checking tool Uppaal. Timed automata are chosen as the basis of the model, since the system is very dependent on correct timing. The theory of timed automata provides a framework to model and verify real-time systems.

The checked system is a safety instrumented system (SIS) that is controlled by a single programmable logic controller (PLC). The purpose of the system is to cut electricity from a protected area, if an electric arc is observed. Because of the complexity of the system, standard testing can not guarantee the correct functioning.

### 1.3 Outline of the Work

The rest of this work is organized as follows. In Section 2 timed automata are introduced and the model checking methodology of this work is presented. In Section 3 the use of programmable logic controllers in timed safety instrumented systems is discussed. A survey of related research is in Section 4. The case study of this work is presented in Section 5, where two different models of the system are shown. The results of the verification of the models are in Section 6. Finally, the conclusions of the work are in Section 7.

## 2 MODEL CHECKING OF TIMED SYSTEMS

Model checking methods often use automata as their primary modelling structure. The automata can be finite state automata, timed automata, Büchi automata or of some other automata class depending on the employed model checking method. Automata provide a way to describe the behaviour of the modelled system efficiently and precisely. Also, the modelling of specifications by automata is possible. This provides a useful model checking approach of a system. The specification automaton and the automaton of the system can be run in parallel. Usually model checking tools create a parallel composition of the system automaton and the negation of the specification

automaton. If the created automaton is not empty, the specification is not met by the system. A counterexample can be easily extracted from the parallel composition.

A finite automaton [20] is a mathematical model of a system that has a constant amount of memory that does not depend on the size of the input. Automata can operate on finite or infinite words depending on definition.

**Definition 2.1 (Finite Automata)** *A finite automaton over finite words $A$ is a five tuple $\langle \Sigma, Q, \Delta, Q^0, F \rangle$ such that*

- *$\Sigma$ is the finite alphabet,*
- *$Q$ is the finite set of states,*
- *$\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation,*
- *$Q^0 \subseteq Q$ is the set of initial states, and*
- *$F \subseteq Q$ is the set of final states.*

Usually automata are depicted as graphs with labeled transitions, where the set of states $Q$ is represented by the nodes and the transition relation $\Delta$ is transformed to the edges of the graph. An example of a finite automaton is in Figure 1.

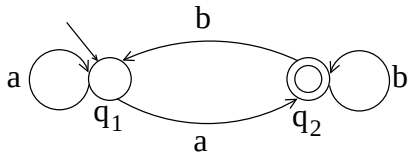


Figure 1: A finite state automaton

In the example automaton $\Sigma = \{a, b\}$, the set of states $Q = \{q1, q2\}$, $Q^0 = \{q1\}$ and $F = \{q2\}$. Initial states are marked with an incoming arrow. Final states are the ones with a double circle. In the example the transition relation is such that $\Delta = \{(q1, a, q1), (q1, a, q2), (q2, b, q2), (q2, b, q1)\}$.

The following definitions follow closely [20]. A word $v$ is a sequence of $\Sigma^*$ whose length is denoted by $|v|$. The $i$th input letter of the word $v$ is denoted by $v(i)$. A run $\rho$ over $v$ is a path in the automaton graph from an initial state $\rho(0)$ to a state $\rho(|v|)$. Formally, a run $\rho$ of an automaton $A$ over $v$ is a mapping $\rho : \{0, 1, ..., |v|\} \mapsto Q$ such that:

- The first state is an initial state, $\rho(0) \in Q^0$.
- Moving from $\rho(i)$ to $\rho(i + 1)$ upon reading the $i$th input letter $v(i)$ is consistent with the transition relation. For every $i$, $0 \leq i < |v|(\rho(i), v(i), \rho(i + 1)) \in \Delta$.

A run $\rho$ over a word $v$ is accepting if it ends in a final state, $\rho(|v|) \in F$. The language of an automaton $A$, $L(A)$ is the set of words accepted by $A$.

## 2.1 Timed Automata

Timed automata [3, 8] are used in the model checking of real time systems. Alternative methods with the same goal are e.g., Petri Nets, timed process algebras, and real time logics [16, 40, 42, 50]. Timed automata are especially needed when the correct functioning depends fundamentally upon real time considerations. Such a situation is typical when the system must interact with a physical process.

Modal logic [21] considers only the ordering of sequential events, i.e., it abstracts away from time. However, in the linear time model an execution of a system can be modelled as a timed trace, in which the events and their actual time points are denoted. The behaviour of a system is a set of these timed traces. A set of timed traces can be thought of as a set of sequences that form a language. If the language is regular, it is possible to use finite automata in the process of specification and verification of the system.

In the original theory [3] timed automata are essentially finite state automata extended with real valued clock variables and infinite input. The functionality of the automaton can be restricted by the conditions set to the clocks.

A timed automaton is an abstraction of a real time system. It is basically a finite state automaton with a set of clock variables. The variables model the logical clocks of the system, and they are initialized with zero when the system is started. After this, all the clock variables are increased at the same rate. In addition to the clocks, a timed automaton also has guard constraints on its transitions. A transition can be taken, when the guard constraint on the edge of the automaton evaluates to true. These guards restrict the behaviour of the automaton by constraining the values of the clocks allowed for the transition to be enabled.

Finally, the clock variables can also be reset. This can only happen when a transition is taken. Multiple clocks can be reset at once. The clock variables are reset after the guard constraint has been evaluated as true.

The problem with the original timed automaton is that the guards only enable the transitions. The automaton can not be forced to make transitions. This leads to a possible situation where the automaton stays forever in some state [8].

A simplified version of a timed automaton, a timed safety automaton [29] is a timed automaton with local state invariants. A timed safety automaton may stay in a node only as long as the clocks satisfy the invariant of the node. These invariant conditions can eliminate the problem because they can force the automaton to make a transition. Because of its simple structure, the timed safety automaton has been adopted in many timed automata verification tools including Uppaal [34] and Kronos [51].

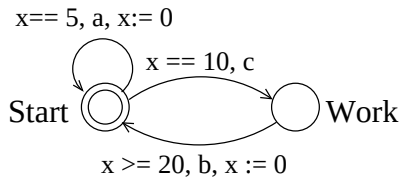


Figure 2: A timed automaton

An example of a timed automaton is in Figure 2. The timed automaton in Figure 2 has two locations: *Start* and *Work*, and a clock variable $x$. *Start* has a double circle surrounding it indicating the initial location insted of the incoming arrow in Figure 1. The automaton has three transitions. Each transition has a guard and an action. There is a transition from *Start* to itself. The guard of this transition states that the transition can only be taken, when the clock variable has value 5 ($x == 5$). The action related to this transition is $a$. The clock is reset after the transition ($x := 0$). The transition from *Start* to *Work* has a guard $x == 10$ and an action $c$. This transition does not reset the clock variable. The third transition from *Work* to *Start* has a guard $x >= 20$ and an action $b$. The guard states that the transition can not be taken unless $x$ is at least 20. The transition also resets the clock $x$. It is also possible to remain in either one of the locations forever. The automaton has no location invariants.

When location invariants are added to the example automaton, the result is a timed safety automaton (Figure 3). It has an invariant in both locations. The invariants specify a local condition that *Start* must be left before $x$ becomes greater than 10, and *Work* must be left before $x$ becomes greater than 50.

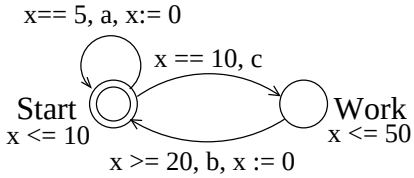


Figure 3: A timed automaton with invariant constraints

This work concentrates on timed safety automata, and will herefrom refer to them as timed automata or automata.

### 2.1.1 Formal Semantics

Basic definitions of the syntax and semantics of timed automata are given. The definitions follow the semantics in [8]. The following notations are used: $\mathbb{N}$ is the set of natural numbers, $C$ is the set of clocks, $B(C)$ is a set of simple conjunctions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C, c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. A timed automaton is a finite graph, with transitions labelled with conditions over and resets of non-negative real valued clock variables.

**Definition 2.2 (Timed Automata)** *A timed automaton $A$ is defined as a tuple $\langle L, l_0, C, \Sigma, E, I \rangle$, where*

- *$L$ is a finite set of locations (or nodes),*
- *$l_0 \in L$ is the initial location,*
- *$C$ is the finite set of clocks,*
- *$\Sigma$ is the finite set of actions,*
- *$E \subseteq L \times \Sigma \times B(C) \times 2^C \times L$ is the finite set of edges between locations with an action, a guard, and a set of clocks to be reset; and*
- *$I : L \longrightarrow B(C)$ assigns invariants to locations.*

Next the semantics of a timed automaton is defined. A clock valuation is a function $u : C \to \mathbb{R}_+$ from the set of clocks to the non-negative reals. Let $\mathbb{R}^C$ be the set of clock valuations. Let $u_0(x) = 0$ for all $x \in C$. In our notation guards and invariants can be considered as sets of clock valuations. $u \in I(l)$ means that the clock valuation $u$ satisfies all the constraints in $I(l)$.

For $d \in \mathbb{R}_+$, let $u + d$ denote the clock assignment that maps all $x \in C$ to $u(x) + d$, and for $r \subseteq C$, let $[r \mapsto 0]u$ denote the clock assignment that maps all clocks in $r$ to 0 and agree with $u$ for the other clocks in $C \setminus r$.

The semantics of a timed automaton is defined as a labelled transition system where a state consists of the current location, and the current values of the clock variables. Thus, there are two types of transitions between states. In a delay transition the automaton delays for some time (denoted $\xrightarrow{d}$, where $d$ is a non-negative real). In an action transition an enabled edge is followed (denoted $\xrightarrow{a}$, where $a$ is an action). Consecutive delay-action transitions can be denoted as $\xrightarrow{d}\xrightarrow{a}$.

**Definition 2.3 (Semantics of Timed Automata)** *Let $(L, l_0, C, \Sigma, E, I)$ be a timed automaton. The semantics is defined as a labelled transition system $\langle S, s_0, \to \rangle$, where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\to \subseteq S \times \{\mathbb{R}_+ \cup \Sigma\} \times S$ is the transition relation such that:*

- *$(l, u) \xrightarrow{d} (l, u + d)$ if $\forall d' : 0 \leq d' \leq d \Longrightarrow u + d' \in I(l)$, and*
- *$(l, u) \xrightarrow{a} (l', u')$ if there exists $e = (l, a, g, r, l') \in E$ such that $u \in g, u' = [r \mapsto 0]u$, and $u' \in I(l)$.*

The transition relation is intuitively such that it allows two kind of transitions. Either all the clock values of the automata are increased by some positive value, or time does not advance at all while an edge of the automaton is taken. In the first case the transition must be allowed by the location invariants. In the second case the transition can only be taken if the guards

evaluate to true, and the invariant constraints are not violated after the transition's reset phase. As an example of the semantics, the timed automata in Figure 3 could have the following reachable states:

$(Start, x = 0) \xrightarrow{5} (Start, x = 5) \xrightarrow{a} (Start, x = 0) \xrightarrow{10} (Start, x = 10) \xrightarrow{c} (Work, x = 10) \xrightarrow{38} (Work, x = 48) \xrightarrow{b} (Start, x = 0)...$

### 2.1.2 Decision Problems in Timed Automata

In model checking, we need to be able to ask questions about the functioning of the automaton used as a model. Operational semantics is the basis for verification of timed automata [8]. An important question to ask about a timed automaton is the reachability of a certain state in the automaton. These kind of questions are used to formalize safety properties of the system. It is also important to know how to compare the functioning of two independent automata. Two main indications of similarity are language inclusion and bisimilarity. Language inclusion means that the set of traces produced by an automaton $A$ is a subset of the set of traces produced by a different automaton $B$. Bisimilarity is a stronger measure of similarity than language inclusion. A formal definition of bisimulation is presented in what follows. Next, some definitions for language inclusion, bisimulation and reachability in timed automata are given. The definitions in [8] are closely followed.

A timed action $(t, a)$ is a pair, where $a \in \Sigma$ is an action performed by the automaton $A$ at time point $t \in \mathbb{R}_+$. The absolute time $t$ is called the time-stamp of the action $a$. A timed trace $\xi = (t_1, a_1)(t_2, a_2)(t_3, a_3)...$ is a sequence of timed actions where $t_i \leq t_{i+1}$ for all $i \geq 1$.

**Definition 2.4** *A Run of a Timed Automaton $A = \langle L, l_0, C, \Sigma, E, I\rangle$ with initial state $\langle l_0, u_0\rangle$ over a timed trace $\xi = (t_1, a_1)(t_2, a_2)(t_3, a_3)...$ is a sequence of transitions: $\langle l_0, u_0\rangle \xrightarrow{d_1}\xrightarrow{a_1} \langle l_1, u_1\rangle \xrightarrow{d_2}\xrightarrow{a_2} \langle l_2, u_2\rangle \cdots$ satisfying the condition $t_i = t_{i-1} + d_i$ for all $i \geq 1$.*
*The timed language $L(A)$ is the set of all timed traces $\xi$ for which there exists a run of $A$ over $\xi$.*

Language inclusion problem is undecidable for timed automata [3]. This is because timed automata are not determinizable in general. If the time stamps of the traces are not taken into consideration, we can define the untimed language $L_{untimed}(A)$ as the set of all traces in the form: $a_1a_2a_3...$ for which there exists a timed trace $\xi = (t_1, a_1)(t_2, a_2)(t_3, a_3)... \in A$. The language inclusion problem for these untimed languages is decidable [3].

It has been shown that timed bisimulation is decidable [15]. Timed bisimulation is introduced for timed process algebras in [50], and can be extended to timed automata [8].

**Definition 2.5 (Bisimulation of Timed Automata)** *A bisimulation $R$ over the states of timed automata $A_1 = \langle L^1, l_0^1, C^1, \Sigma, E^1, I^1\rangle$ and $A_2 = \langle L^2, l_0^2, C^2,$*

$\Sigma, E^2, I^2\rangle$ *is a symmetrical binary relation satisfying the following condition:*

*for all* $(s_1, s_2) \in R$*,*

*if* $s_1 \xrightarrow{\sigma} s_1' \in E^1$ *for some* $\sigma \in \Sigma$ *and* $s_1, s_1' \in L^1$*, then* $s_2 \xrightarrow{\sigma} s_2' \in E^2$ *and* $(s_1', s_2') \in R$ *for some* $s_2, s_2' \in L^2$*.*

*if* $s_2 \xrightarrow{\sigma} s_2' \in E^2$ *for some* $\sigma \in \Sigma$ *and* $s_2, s_2' \in L^2$*, then* $s_1 \xrightarrow{\sigma} s_1' \in E^1$ *and* $(s_1', s_2') \in R$ *for some* $s_1, s_1' \in L^1$*.*

*Two automata are timed bisimilar iff there is a bisimulation containing the initial states of the automata.*

In the case of bisimulation, an untimed version is also decidable [35]. We just consider a timed transition $s_1 \xrightarrow{d} s_2$ as an empty transition $s_1 \xrightarrow{\varepsilon} s_2$. The alphabet of the automaton is the replaced with $\Sigma \cup \{\varepsilon\}$.

**Definition 2.6 (Reachability Analysis of Timed Automata)**
*Let* $\langle l, u\rangle \rightarrow \langle l', u'\rangle$ *if* $\langle l, u\rangle \xrightarrow{\sigma} \langle l', u'\rangle$ *for some* $\sigma \in \Sigma \cup \mathbb{R}_+$*. Let* $\rightarrow^*$ *denote* $n$ *consecutive transitions, where* $n \in \mathbb{N}$*. For an automaton with initial state* $\langle l_0, u_0\rangle$*,* $\langle l, u\rangle$ *is reachable iff* $\langle l_0, u_0\rangle \rightarrow^* \langle l, u\rangle$*. More generally, given a constraint* $\phi \in B(C)$ *we say that the configuration* $\langle l, \phi\rangle$ *is reachable if* $\langle l, u\rangle$ *is reachable for some* $u$ *satisfying* $\phi$*.*

Reachability analysis offers a lot of model checking properties. Negations of reachability properties can be used to express invariant properties. For example, a system is always in a safe state if the failure states of the system are not reachable. In addition, reachability analysis of timed automata offers a way to examine bounded liveness properties. These properties state that some state will be reached within a given time. The property can be transformed into an invariant property using an additional automaton.

### 2.1.3 Parallel Composition of Timed Automata

A parallel composition of timed automata [3] is an operation used to describe complex systems using simpler subsystems. A parallel composition describes the joint functioning of several automata concurrently.

In an untimed version of the parallel composition, it can be defined using the traces of the automata. An untimed automaton is totally determined by the set of its traces. A parallel composition of these trace sets is the set of traces such that for each automaton the relevant projection is possible in the automaton. If the event sets of the automata are distinct, the parallel composition is just the union of the trace sets. If the event sets of the automata are identical, the parallel composition is the set theoretic intersection of the trace sets.

Next, the parallel composition of timed automata is defined. The definitions in [3] are closely followed. The projection of an untimed trace $\xi = a_1 a_2 a_3 ...$ onto an automaton $A_i$, written $\xi \lceil A_i$ is formed by taking only

the events of the trace $\xi$ that are in the event set of the automaton $A_i$. The projection is only considered when the intersection $\xi \cap A_i$ is nonempty. The parallel composition $\|_i A_i$ for a set of untimed automata $A_i$ is thus an untimed automaton with the event set of $\cup_i A_i$. The trace set of the parallel composition is the set of traces that exist in at least one of the component automata, and can be projected to all of the component automata.

The parallel composition operator can be extended to timed automata as well. The projection operator is changed so that in the parallel composition of two processes the common events should always happen at the same time. A composition of two traces with common events will always result in either an empty set or a single trace.
If, for example, automaton $A_1$ with an event set $\{a, b\}$ has only a single trace

$$\xi_1 = (a,1)(b,2)(a,4)(b,5)(a,7)(b,8)...$$

and an automaton $A_2$ with an event set $\{a, c\}$ has three possible traces:

$$\xi_2 = (a,1)(a,4)(a,7)...$$
$$\xi_3 = (a,1)(a,2)(a,3)...$$
$$\xi_4 = (c,3)(c,6)(c,9)...$$

The resulting parallel composition $A_1 \| A_2$ would have an event set of $\{a, b, c\}$ and a set of traces:

$$\xi_{C1} = (a,1)(b,2)(a,4)(b,5)(a,7)(b,8)...$$
$$\xi_{C2} = (a,1)(b,2)(c,3)(a,4)(b,5)(c,6)(a,7)(b,8)...$$

These are the compositions of trace pairs $(\xi_1, \xi_2)$ and $(\xi_1, \xi_4)$. The trace pair $(\xi_1, \xi_3)$ results in an empty trace because the common event $a$ takes place at different time stamps in the traces.

Following the definitions in [20], the actual timed automaton that represents the parallel composition of two automata $A_1 = \langle L_1, l_0^1, C_1, \Sigma_1, E_1, I_1 \rangle$ and $A_2 = \langle L_2, l_0^2, C_2, \Sigma_2, E_2, I_2 \rangle$ is the timed automaton:

$$A_1 \| A_2 = \langle L_1 \times L_2, l_0^1 \times l_0^2, C_1 \cup C_2, \Sigma_1 \cup \Sigma_2, E, I \rangle$$

where $I(s_1, s_2) = I_1(s_1) \wedge I_2(s_2)$ and the transition relation E is given by the following rules:

- For $a \in \Sigma_1 \cup \Sigma_2$, if $\langle s_1, a, \phi_1, \lambda_1, s_1' \rangle \in E_1$ and $\langle s_2, a, \phi_2, \lambda_2, s_2' \rangle \in E_2$, then $E$ will contain the transition $\langle (s_1, s_2), a, \phi_1 \wedge \phi_2, \lambda_1 \cup \lambda_2, (s_1', s_2') \rangle$.

- For $a \in \Sigma_1 - \Sigma_2$ if $\langle s, a, \phi, \lambda, s' \rangle \in E_1$ and $t \in L_2$, then $E$ will contain the transition $\langle (s,t), a, \phi, \lambda, (s',t) \rangle$.

- For $a \in \Sigma_2 - \Sigma_1$ if $\langle s, a, \phi, \lambda, s' \rangle \in E_2$ and $t \in L_1$, then $E$ will contain the transition $\langle (t,s), a, \phi, \lambda, (t,s') \rangle$.

The locations of the parallel composition automaton are pairs of locations from the component automata. Invariants are conjunctions of the invariants in the component automata. For each pair of transitions from the component automata with the same action, there will be a transition in the composite automaton. The transition source state is a pair in the composition that consists of the source states of the individual automata. The transition target location is such a pair that is formed from the target locations of the individual transitions. If an action only exists in one of the automata, the composition transition will be such that the other automaton remains unchanged. Such a transition is created for each location of the other automaton.

### 2.1.4 Symbolic Semantics, Regions and Zones

A timed automaton with real-valued clocks leads to an infinite transition system. In order to perform efficient verification of timed automata, a finite transition system must be acquired. The basis of decidability results in timed automata comes from the concept of region equivalence over clock assignments [3]. The next section follows closely the definitions in [8].

**Definition 2.7 (Region Equivalence)** *Let $k$ be a function, called a clock ceiling, mapping each clock $x \in C$ to a natural number $k(x)$ (i.e. the ceiling of $x$). For a real number $d$, let $\{d\}$ denote the fractional part of $d$, and let $\lfloor d \rfloor$ denote its integer part. Two clock assignments $u, v$ are region-equivalent, denoted $u \stackrel{\cdot}{\sim}_k v$, iff*

- *for all $x$, either $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or both $u(x) > k(x)$ and $v(x) > k(x)$,*
- *for all $x$, if $u(x) \leq k(x)$ then $\{u(x)\} = 0$ iff $\{v(x)\} = 0$ ; and*
- *for all $x, y$ if $u(x) \leq k(x)$ and $u(y) \leq k(y)$ then $\{u(x)\} \leq \{u(y)\}$ iff $\{v(x)\} \leq \{v(y)\}$.*

A region is an equivalence class denoted $[u]$ that is the set of region-equivalent clock assignments with $u$. Using the region construction, a finite partitioning of the state space is possible. This is because each clock has a maximal constant value $k(x)$ which makes the number of regions finite. The constant value $k(x)$ is the highest value, against which the clock is compared.

Also, $u \stackrel{\cdot}{\sim} v$ implies that the states of the timed automaton $(l, u)$ and $(l, v)$ are bisimilar with regard to the untimed bisimulation for any location $l \in L$. The equivalence classes can be used to create a finite-state region automaton. Using a region automaton, many of the decision problems of timed automata become decidable. The transition relation between symbolic states of a region automaton is the following:

- $\langle l, [u] \rangle \Rightarrow \langle l, [v] \rangle$ if $\langle l, u \rangle \xrightarrow{d} \langle l, v \rangle$ for a positive real number $d$, and
- $\langle l, [u] \rangle \Rightarrow \langle l', [v] \rangle$ if $\langle l, u \rangle \xrightarrow{a} \langle l', v \rangle$ for an action $a$.

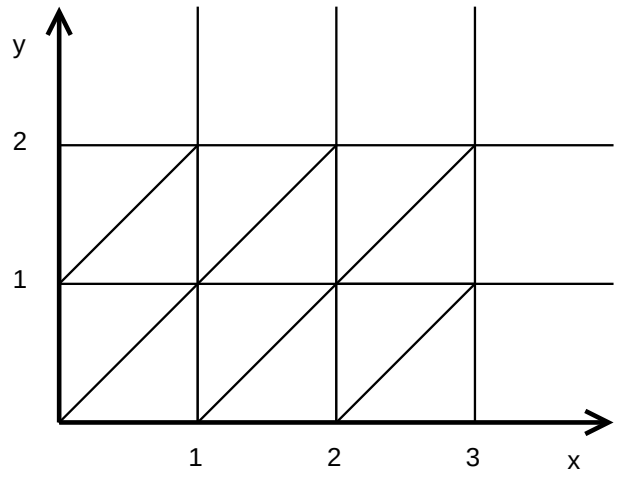


Figure 4: Regions of a system

An example of the regions of an automaton with two clocks $x$ and $y$ is in Figure 4. The maximal comparison constants of $x$ and $y$ are 3 and 2, respectively. The example has 60 different time regions. All open areas, lines and intersections count as a region. Possible regions of the example are $(x = 1, y = 1)$ (a corner point), $(x = 2, y < 1)$ (a line segment), $\{(1 < x < 2) \wedge (y < x)\}$ (an open area).

The intuitive idea of using regions is the following: if two states, which correspond to the same location of a timed automaton, have clock values with the same integral parts and the ordering of the fractional parts, the two states will behave similarly.

The problem of the region automata is the exponential growth in the number of regions as the number of clocks or the maximal constants increase. Clock zones [1] can represent the state space of a timed automaton more efficiently. [19, 29]

The idea of clock zones is that most of the time regions are not needed, and some of them can be united. A clock zone is a set of clock assignments i.e. a conjunction of inequalities or a convex union of clock regions, that compare a clock value or the difference between two clock values against an integer. The following types of inequalities are allowed:

$$x < c, x \leq c, c < x, c \leq x, x - y < c, x - y \leq c$$

where $c$ is an integer, $x, y$ are clocks. For a clock zone $\phi$, the set of clock values satisfying $\phi$ will also be denoted $\phi$. If an automaton $A$ has $k$ clocks, then a clock zone $\phi$ expressed in terms of these clocks is a convex subset in k-dimensional Euclidean space [20].

For example, one possible zone graph of the timed automaton in Figure 3 is in Figure 5.

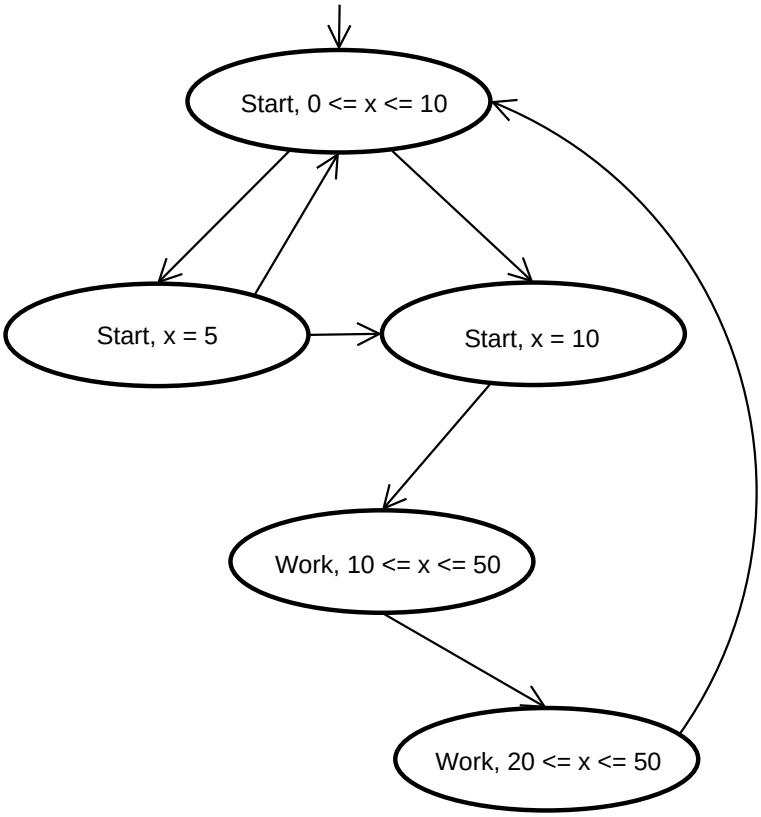


Figure 5: The zone graph of the timed automaton in Figure 3

### 2.1.5 Difference Bound Matrices

Difference bound matrix (DBM) [19] is a way to represent a clock zone in a compact form. We define a difference bound matrix following the definition in [20]. Its definition requires the use of a special clock $c_0$ that always has value 0. The difference bounded matrix is indexed by the set of clocks $C_0 = C \cup \{c_0\}$. The special clock $c_0$ has the index 0. The entries of the matrix $D_{i,j}$ have the form $(d_{i,j}, \prec_{i,j})$ that expresses a comparison of two clock values $c_i$ and $c_j$ with an integer $d_{i,j}$. The comparison operator $\prec_{i,j}$ is either $<$ or $\leq$. The matrix entries represent inequalities $c_i - c_j \prec d_{i,j}$, where $d_{i,j}$ is either integer or $\infty$. The special clock $c_0$ can be used to represent inequalities that only concern one clock variable. As an example, consider the following clock zone:

$$c_2 - c_1 < -2 \wedge c_2 \leq 1 \wedge c_1 \leq 3$$

The difference bounded matrix is:

$$D = \begin{pmatrix} (0,\leq) & (0,\leq) & (0,\leq) \\ (3,\leq) & (0,\leq) & (\infty,<) \\ (1,\leq) & (-2,<) & (0,\leq) \end{pmatrix}$$

A zone can be represented by $\mid C_0 \mid^2$ atomic constraints of the form $c_1 - c_2 \prec n$. Each pair is used only once. In the case of two constraints on the same pair of variables, the intersection of these constraints is mean-

ingful. These zones can be stored in $\mid C_0 \mid \times \mid C_0 \mid$ sized matrices called difference bound matrices.

The zone representation is not unique. The same zone can be represented by several different matrices. In our example $c_1 - c_0 \leq 3$ and $c_0 - c_2 \leq 0$ implies $c_1 - c_2 \leq 3$. We can change $D_{1,2}$ to $(3, \leq)$ and obtain an alternative DBM. Generally, the sum of the upper bounds $c_i - c_j$ and $c_j - c_k$ is an upper bound on the clock difference $c_i - c_k$. Reducing the clock differences to tighten the difference bound matrix is done as follows:

If $c_i - c_j \prec_{i,j} d_{i,j}$ and $c_j - c_k \prec_{j,k} d_{j,k}$, then $c_i - c_k \prec'_{i,k} d'_{i,k}$ where $d'_{i,k} = d_{i,j} + d_{j,k}$ and

$$\prec'_{i,k} = \begin{cases} \leq & \text{if } \prec_{i,j} = \leq \text{ and } \prec_{j,k} = \leq \\ < & \text{otherwise} \end{cases}$$

If $(d'_{i,k}, \prec'_{i,k})$ is a tighter bound than $(d_{i,k}, \prec_{i,k})$, the original bound can be replaced by the new one. The operation is called tightening. The DBM is in a canonical form when no further tightening is possible. The canonical form of the DBM in our example is:

$$D = \begin{pmatrix} (0, \leq) & (-2, <) & (1, <) \\ (3, \leq) & (0, \leq) & (3, \leq) \\ (1, <) & (-2, <) & (0, \leq) \end{pmatrix}$$

## 2.2 Temporal Logic with Real Time

Temporal logic is an extension of classical logic that can be used to create formal system specifications. These formal specifications can then be checked using some model checking method. With temporal logic unambiguous descriptions such as "The system never reaches an erroneous state." or "This action always leads to the resetting of the system." can be written.

Temporal logics can be classified according to the assumed structure of time. Some temporal logics assume linear time structure, some assume a branching time structure. Computation tree logic (CTL) [20] is a branching time logic. It is used when the models that are verified are finite state systems that abstract away from time. It is assumed that an execution can be modelled as a linear sequence of system events.

Timed computation tree logic (TCTL) [2] is an extension of CTL to real time systems. For real time systems ordinary CTL is not sufficient, since a system's correctness depends on the values of the timing delays. Sometimes it is not enough if a function is known to eventually happen. In real time systems we need to know whether the action takes place within a certain time period.

In order to create real time models and specifications, using event sequences is not sufficient and therefore timed traces are needed. Timed traces

associate with each state the time of the occurrence of the event. The concept of time can be modelled in different ways. In the case of timed computation tree logic a dense-time model is preferred. In a dense-time model the times of events are real numbers, that increase monotonically without a bound [3]. TCTL was created to describe CTL specifications in real time.

In TCTL, quantitative temporal operators are introduced to describe timed properties. First, the syntax and semantics of the branching-time logic CTL are reviewed. Next, the TCTL extensions to the CTL syntax are defined. TCTL semantics is also represented.

### 2.2.1 Computation Tree Logic

In CTL time is seen as a tree-like structure in which the future in not determined. Different possible futures exist, and any of these is possible. The following section follows the notations in [20].

CTL formulas consist of logical operators, path quantifiers and temporal operators. Path quantifiers ($A$("for all computation paths") and $E$("for some computation path")) are used in a state to specify that all of the paths ($A$) or some of the paths ($E$) starting from that state have some property. The temporal operators describe properties of a path of the tree. Several temporal operators exist. Here, only some are defined, since others can be defined using them.

- **X** ("next") requires that the property holds at the next state of the path.
- **U** ("until") is a binary operator. Formula $P$ **U** $Q$ holds when $P$ is true until $Q$ becomes true. Also, the second argument must become true at some point.

Given a finite set of atomic propositions $\{AP\}$, the CTL formulas can be inductively defined as follows:

$$\begin{aligned} \phi \quad ::= \quad & p \mid \mathit{false} \mid \phi_1 \rightarrow \phi_2 \mid \\ & \mid \mathbf{EX}\phi_1 \mid \mathbf{E}\,[\phi_1 \mathbf{U}\, \phi_2] \mid \mathbf{A}\,[\phi_1 \mathbf{U}\, \phi_2] \end{aligned}$$

where $p \in AP$ is an atomic proposition and $\phi_1, \phi_2$ are CTL formulas. $\mathbf{EX}\phi_1$ means that there is an immediate successor state that is reachable in one step, in which $\phi_1$ is true. $\mathbf{E}\,[\phi_1 \mathbf{U}\, \phi_2]$ requires that there is a path in which $\phi_2$ becomes true at some time point $t$. Also, $\phi_1$ must be true on that path until $t$. $\mathbf{A}\,[\phi_1 \mathbf{U}\, \phi_2]$ means that for every computation path, the previous condition holds.

Other often used temporal operators are for example: $\mathbf{EF}\phi$ for $\mathbf{E}\,[true \mathbf{U}\, \phi]$, $\mathbf{AF}\phi$ for $\mathbf{A}\,[true \mathbf{U}\, \phi]$, $\mathbf{EG}\phi$ for $\neg\mathbf{AF}\neg\phi$ and $\mathbf{AG}\phi$ for $\neg\mathbf{EF}\neg\phi$.

The semantics of CTL is defined with respect to a Kripke structure $M = \langle S, R, L\rangle$, where $S$ is the set of states, $R \subseteq S \times S$ is the total transition relation, and $L : S \rightarrow 2^{AP}$ is the labelling function. A path in $M$ is an infinite

sequence of states, $\pi = s_0, s_1, s_2, ...$ such that for every $i \geq 0, (s_i, s_{i+1}) \in R$. We denote with $\pi^i$ the suffix of $\pi$ starting at $s_i$. Notation $M, s \models f$ means that $f$ holds at a state $s$ in a structure $M$. Let $Tr(s) = \{\pi = s_0, s_1, ... \mid s_0 = s\}$ be the set of possible paths starting from the state $s$. The satisfaction relation $\models$ is defined inductively:

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

### 2.2.2 Timed Computation Tree Logic

It is possible to write properties like $\mathbf{EF}p$ in CTL. However, CTL does not provide a way to bound the time at which $p$ happens. TCTL extends the temporal operators so that it is possible to limit their scope in time. It is possible to write for example $\mathbf{EF}_{<5}p$ meaning that at some computation path $p$ will become true within 5 time units. TCTL syntax is shortly:

$$
\begin{aligned}
\phi \quad ::= \quad & p \mid false \mid \phi_1 \rightarrow \phi_2 \mid \\
& \mid \mathbf{E}\,[\phi_1 \mathbf{U}_{\sim c}\, \phi_2] \mid \mathbf{A}\,[\phi_1 \mathbf{U}_{\sim c}\, \phi_2]
\end{aligned}
$$

where $p \in AP$ is an atomic proposition, $c \in N$, $\phi_1$ and $\phi_2$ are TCTL formulas and $\sim \in \{<, \leq, =, \geq, >\}$.

$\mathbf{E}\,[\phi_1 \mathbf{U}_{<c}\, \phi_2]$ means that for some computation path there exists a prefix of time length less than $c$ time steps, such that at the last state of the prefix $\phi_2$ holds, and $\phi_1$ is true in all the states in the path until the last state. $\mathbf{A}\,[\phi_1 \mathbf{U}_{<c}\, \phi_2]$ means that the above condition holds on every computation path. It is also possible to create TCTL formulas for time intervals. For example a formula $\mathbf{EF}_{(a,b)}\phi$ meaning "$\phi$ holds at least once between time steps $a$ and $b$" can be written $\mathbf{EF}_{=a}\mathbf{EF}_{<(b-a)}\phi$.

Since TCTL operates in a dense time domain and not in a discrete time domain like CTL, the next-time operator can not be used. By definition, there is no unique next time point. The computation paths in TCTL with dense time domain are maps from the real valued time domain $R$ to states of the system. There is a unique state at every real valued time instant. For a set of states $S$ and a state $s \in S$ an $s$-path through $S$ is a map $p$ from $R$ to $S$ satisfying $p(0) = s$. The computation tree in dense time is a map from every

state to a set of paths starting at that state. The prefix of an $s$-path up to time $t$ is denoted $p_t$. The concatenation of two $s$-paths $p_1$ and $p_2$ is denoted $p_1 \centerdot p_2$.

The structure that TCTL can be defined against can not be exactly the same as in the of CTL. The TCTL structure is a triple $M = \langle S, f, L \rangle$ where $S$ is the set of states, $L : S \rightarrow 2^{AP}$ is the labelling function, and $f$ is a map giving for each $s \in S$ a set of s-paths through $S$. $f$ satisfies the tree constraint:

$$\forall s \in S. \forall p \in f(s). \forall t \in R. p_t \centerdot f[p(t)] \subseteq f(s).$$

The satisfaction relation $\models$ for TCTL is defined inductively:

$$\begin{array}{rll}
M, s \models p & \textit{iff} & p \in L(s) \\
M, s \models \neg\phi & \textit{iff} & M, s \not\models \phi \\
M, s \models \phi_1 \rightarrow \phi_2 & \textit{iff} & M, s \not\models \phi_1 \text{ or } M, s \models \phi_2 \\
M, s \models A[\phi_1 U_{\sim c} \phi_2] & \textit{iff} & \forall p \in f(s) : \exists t \sim c, p(t) \models \phi_2 \wedge \\
 & & (\forall (0 < t' < t) p(t') \models \phi_1) \\
M, s \models E[\phi_1 U_{\sim c} \phi_2] & \textit{iff} & \exists p \in f(s) : \exists t \sim c, p(t) \models \phi_2 \wedge \\
 & & (\forall (0 < t' < t) p(t') \models \phi_1)
\end{array}$$

## 2.3 Model Checking Tool Uppaal

Uppaal is a tool for model checking of timed systems. Other tools for modelling and verification based on timed automata are e.g., Kronos [51] and RED [48].

The Uppaal modelling language [34] is based on networks of timed automata. A network of timed automata is a parallel composition $A_1 \mid \cdots \mid A_n$ of timed automata $A_1, ...A_n$, referred to as processes. The definition of a parallel composition varies depending on the used process calculi. In Uppaal the parallel composition operator of Calculus of Communicating Systems (CCS) [39] is used. Hand-shake synchronization with input and output actions is used for synchronous communication. Asynchronous communication happens through shared variables. For hand-shake synchronization purposes the action alphabet $\Sigma$ in Uppaal consist of symbols $a?$ for input actions and $a!$ for output actions. Internal actions are represented by a distinct symbol $\tau$. Next we define a network of timed automata formally.

A network of timed automata has a common set of clocks and actions. The network consists of $n$ timed automata $A_i = (L_i, l_i^0, C, \Sigma, E_i, I_i), 1 \leq i \leq n$. A location vector is $\overline{l} = (l_1, ..., l_n)$. We write $I(\overline{l}) = \bigwedge_i I_i(l_i)$ as a composition of the invariant functions. Let $\overline{l}[l_i'/l_i]$ mark the vector $\overline{l}$ with the $i$th element $l_i$ replaced by $l_i'$.

**Definition 2.8 (Semantics of a network Timed Automata)** *Let* $A_i = (L_i, l_i^0, C, \Sigma, E_i, I_i), 1 \leq i \leq n.$ *be a network of timed automata. Let* $\overline{l}_0 =$

*$(l_1^0, ..., l_n^0)$ be the initial location vector. The semantics is defined as a labelled transition system $\langle S, s_0, \rightarrow \rangle$, where $S = (L_1 \times \cdots \times L_n) \times \mathbb{R}^C$ is the set of states, $s_0 = (\bar{l}_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times \{\mathbb{R}_+ \cup \Sigma\} \times S$ is the transition relation defined by:*

- $(\bar{l}, u) \rightarrow (\bar{l}, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \Longrightarrow u + d' \in I(\bar{l})$
- $(\bar{l}, u) \rightarrow (\bar{l}[l'_i/l_i], u')$ *if there exists* $l_i \xrightarrow{agr} l'_i$ *such that* $u \in g, u' = [r \mapsto 0]u$, *and* $u' \in I(\bar{l})$ *; and*
- $(\bar{l}, u) \rightarrow (\bar{l}[l'_j/l_j, l'_i/l_i], u')$ *if there exists* $l_i \xrightarrow{c?g_i r_i} l'_i$ *and* $l_j \xrightarrow{c!g_j r_j} l'_j$ *such that* $u \in (g_i \wedge g_j), u' = [r_i \cup r_j \mapsto 0]u$ *and* $u' \in I(\bar{l})$.

### 2.3.1 Modelling in Uppaal

Modelling in Uppaal is done via a graphical user interface. Timed automaton templates can be created with the Uppaal modelling tool. Every template has its own local declaration section, where local variables and functions can be introduced. There is also a global declarations' section for global variables and functions. Finally, a section for process declaration is needed. In this part the automaton instances are created from the templates, and the parallel composition of these is declared for property checking and simulation.

In addition to creating networks of timed automata, the Uppaal modelling language is extended with several modelling features:

- Templates of automata can be defined with a set of parameters. The parameters are substituted in the process declaration part.
- Integer constants can be defined.
- Bounded integer variables (int[min, max]) can be defined.
- Binary synchronization channels can be declared.
- Broadcast channels can be declared. In broadcast synchronization one transition labelled with an output action can synchronize with several transitions labelled with an input action.
- Synchronization channels can be declared urgent by prefixing the channel declaration with the keyword urgent. When a transition labelled with an urgent synchronization is enabled (i.e., it can be taken), time is not allowed to pass. However, the synchronization transition need not be taken if other transitions are possible.
- Locations can be declared urgent. When a system is in an urgent location, no time is allowed to pass.
- Locations can be declared committed. A state is committed if one or more locations in the state are committed. When a system state is committed, no time is allowed to pass. Also, the next transition must be such that an outgoing edge of at least one of the committed locations is involved.

- Arrays of clocks, channels, constants or integers can be declared.
- Integer variables and arrays can be initialized.

### 2.3.2 Verification in Uppaal

As specifications, Uppaal accepts a subset of TCTL formulas. In general, Uppaal does not allow nesting of temporal operators, or bounded specifications. In Uppaal syntax [] is equivalent to the TCTL **G**, or "globally". $\langle\rangle$ is equivalent to the TCTL **F**, or "finally". The notation $p --> q$ (p leads to q) is an abbreviation of $A[](p \; imply \; A <> q)$.

In Uppaal specifications, the dot character (.) is used to reference the variables and states of a particular timed automaton. For example $A.l$ references to the location or variable $l$ of automaton $A$.

In the verification process, Uppaal uses symbolic states of the timed automata $\langle l, D\rangle$ where $l$ is a location of the automaton, and $D$ is a zone stored in memory as a DBM [5]. The Uppaal tool calculates the parallel composition of the timed automata in the model, and performs reachability analysis on the structure in order to verify it against a property. In other words, the tool goes through the state space, and tries to find a state, in which the property is false.

Not all specifications can be stated in TCTL as supported by Uppaal. In these cases some modelling tricks can be used. Often it is possible to create an additional observer automaton that observes the behaviour of some other automata. For instance, bounded liveness properties of TCTL can be checked using an observer automaton.

An example of an observer automaton is in Figure 6. Using the observer automaton, a bounded liveness property:

**AG** $alarm \; imply$ **AF** $_{<50} \; observation$

that can not be checked as such with Uppaal, can be stated differently as:

```
A[] not Observer.Failure
```

Observer automata can be used in many ways and with various properties.

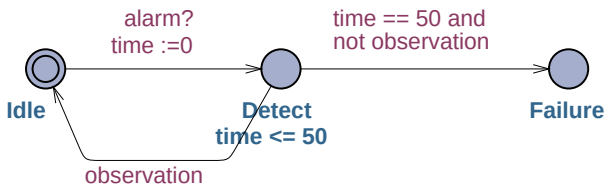


Figure 6: An observer automaton in Uppaal

# 3 TIMED SAFETY INSTRUMENTED SYSTEMS

## 3.1 Programmable Logic Controller

A programmable logic controller (PLC) defined in IEC 1131-3 is a digital computer that can be used in automation control and safety instrumentation control.

PLCs have evolved from simple logic controllers used to control physical processes that have a number of inputs, outputs, relays and timers. PLCs were designed as a replacement to logic controllers with relays [23]. PLCs, however, can be modified to work like different logic controllers. The IEC standard describes five different programming language standards for PLCs:

- Ladder diagram (LD),
- Function block diagram (FBD),
- Structured text (ST),
- Instruction list (IL), and
- Sequential function chart (SFC).

Structured text and instruction list are textual PLC programming languages. The other three are graphical diagram based languages. PLCs support complex features such as multi-tasking and interrupts, but these are not necessary, and will only make the verification difficult. Therefore, a simple version of a PLC is used throughout this work. The characterization follows.

A PLC program uses two memory areas reserved for input and output signal values. Before each execution cycle the sensors of the PLC are polled and the values are copied to the memory area reserved for the input signals. This part of memory contains the snapshot of the input values at the time of the polling. After the PLC program has been executed, the output values are updated. A well written PLC program terminates within a bounded amount of time which is less than the cycle time of the PLC. The PLC program will initiate the next cycle after some fixed amount of time.

Timers are pieces of programs used with systems that need real-time features. The timers used in our simple version of a PLC are of type TON (Timer On Delay). A TON timer has some signal as an input $IN$, a preset integer value $PT$, an internal accumulator variable, timers base interval value, and two output signals $Q$ and $ET$. The timer records the number of base intervals the input signal has been true, and increases the accumulator value accordingly. When the accumulator value is greater than or equal to the preset value $PT$, the output signal $Q$ is turned on. $ET$ is an integer output that has an initial value of 0. In every cycle the value of $ET$ is increased by 1, if the $IN$ is true, and the current value of $ET$ is less than $PT$. If $IN$ is true, and $ET$ is not less than $PT$, the value of $ET$ is not increased. When the input $IN$ is false, both outputs are reset to zero. The preset value $PT$ can

also be seen as an integer input. A TON timer is represented in Figure 7.

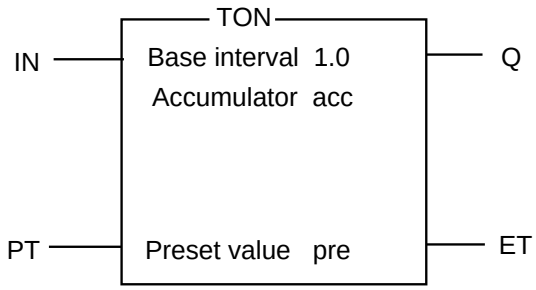


Figure 7: A TON timer with inputs IN and PT, and outputs Q and ET

The functionality of a TON timer is in Figure 8.

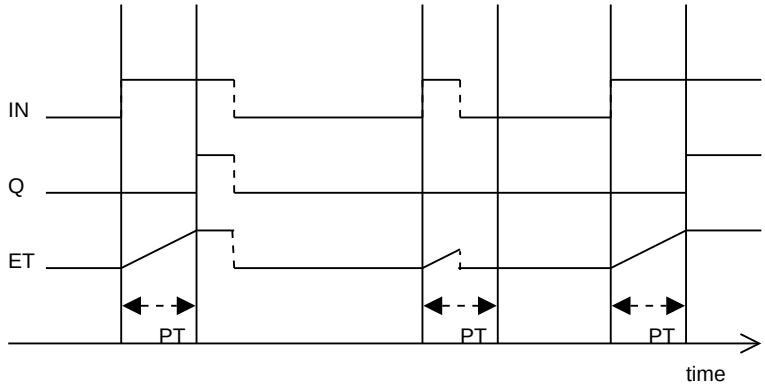


Figure 8: Functionality of a TON timer

## 3.2 Safety Instrumented Systems

Industrial processes involve great risks because of dangerous temperatures, pressures and materials. Therefore, separate systems to protect environment, personnel and equipment are needed. In the ANSI/ISA-84.00.01-2004 (IEC 61511) standard a safety instrumented system (SIS) [23] is defined as an "instrumented system used to implement one or more safety instrumented functions. A SIS is composed of any combination of sensor(s), logic solver(s), and final element(s)". The purpose of a SIS is to either automatically take an industrial process to a safe state, when predetermined conditions are violated; allow a process to move forward when the predetermined conditions are true; or mitigate the consequences of an industrial hazard. A SIS is designed to always work in a risk reducing manner. [23]

The sensors of a SIS collect information of the state of the process. Sensors can measure temperature, pressure, flow or other process parameters. The logic solver makes decisions of the actions taken based on the sensors' signals. A typical action is a signal sent to the final elements. Final elements are usually valves or electrical switches that have some risk reducing effect

on the process.

An example of a SIS is an emergency cooling system of a reactor. The SIS has heat sensors, and a logic, which determines when the safety instrumented function is initiated. In this case the safety instrumented function is the opening of a coolant valve. Similarly, a SIS observing the pressure of a tank, initiates an open action of a pressure releasing valve.

As mentioned earlier, PLCs can be used as the logic solvers of safety instrumented systems. Since, PLCs are increasingly used in safety critical systems, testing and verification of PLC applications has become very important [23].

# 4 MODELLING SYSTEMS WITH TIMED AUTOMATA

In this section, some research in the area of model checking with timed automata is surveyed. The research can be roughly divided into model checking of real-time communication protocols, and model checking of real-time controllers. Most of the surveyed case studies use the model checking tool Uppaal. A comprehensive tutorial on Uppaal is in [6]. In this paper the tool itself and its use is described. In addition, two extensive examples and some modelling conventions are given.

## 4.1 Modelling Real-Time Communication Protocols

Protocol verification has been of interest to many research groups. Network and other communication protocols have been modelled and verified using Uppaal, and other modelling tools. Several protocols that have been commonly in use have been found erroneous, and corrected using Uppaal.

In [44] and [43] a fault tolerant clock synchronization mechanism for a Controller Area Network (CAN) was modelled and verified. The modelling was done using the Uppaal tool. The goal of the work was to formally verify the precision that could be achieved, and the effects of faults to the precision. In their system, master nodes regularly transfer clock synchronization messages to the slave nodes. As an essential part of this research, clock drifting was modelled using clock automata where the clocks operated in variable length cycles. In their model the clock rate could change dynamically. The clock synchronization system corrected both the offset and the drift error of the clocks. A certain precision was verified using an observer automaton that compared the clocks of the nodes.

In [7] the correctness of the Philips Audio Control Protocol was verified using the Uppaal tool. The analysis was performed on a system with two senders. Consequently, the bus collision problem was present. In addition, for an incorrect implementation of the protocol, a counterexample was found. An important observation of the paper was the usefulness of the com-

mitted locations in Uppaal. The Uppaal tool was extended to include these features. In this research, clocks with drifting timespeeds were used. The system was verified for an error tolerance of 5 % on the timing.

In [33] a Collision Avoidance Protocol was studied. The protocol was modelled and verified using two tools SPIN [31] and Uppaal. The timed aspects were easy to model with Uppaal. It was also noticed that the notion of committed locations in Uppaal supported the modelling of broadcast communication, and yielded significant reductions in time- and space-usages.

An Audio/Video protocol by Bang & Olufsen (B&O) was studied in [27]. The protocol controls the transmission of messages over a single bus, and detects collisions. The protocol was known to be faulty, although the cause of the fault could not have been pinpointed before. Using the Uppaal tool, an error trace was extracted. This led to the detection of the error in the implementation. The corrected implementation was successfully verified.

As a continuation to [27] a different Power Down protocol by Bang & Olufsen was studied in [26]. The modelling of the system resulted in the discovery of three design errors that were identified and corrected. In this paper, modelling techniques for time slicing problems with interruptions are introduced. In addition, three observer automaton techniques for property verification are introduced.

A bounded retransmission protocol is studied in [18]. In the paper a file transfer service is first specified by stating several logical properties. The bounded retransmission protocol's conformance to these properties is then checked using Uppaal and SPIN. The timed properties are checked using Uppaal.

The Pragmatic General Multicast (PGM) protocol is analyzed in [12]. The protocol intends to guarantee a reliability property: *"a receiver either receives all data packets from transmissions and repairs or is able to detect unrecoverable data packet loss"* A simplified model of the protocol is built and two reliability properties are checked with Uppaal. The properties were verified only with some parameter values, which the paper presents.

In [36] a method for modelling and verifying of the LEGO RCX programs is introduced. A tool is developed, which can automatically translate RCX programs into Uppaal models. Also, a system of two RCX units communicating through an infrared channel is built. In their research their IR protocol and Fischer's mutual exclusion protocol are verified. Their experiments with an actual system of two communicating RCX units showed that an Uppaal model could be created using their tool, but the model could not be verified because of its complexity.

## 4.2 Modelling Real-Time Controllers

Controllers have also been of interest to many researchers. Programmable logic controllers (PLCs) that are also analyzed in this work are analyzed in [32], [49], [4] and [38].

In [32] software analysis techniques for PLCs are developed. Various verification techniques are considered for two PLC programming languages: instruction list (IL) and sequential function charts (SFCs). In this work a formal semantics is created for both languages. A model checking method for the untimed version of a SFC program is presented. The analysis techniques are also implemented in industrial size case studies.

In [49] a method is developed to analyze simple PLCs written in the instruction list (IL) language. The PLCs are converted into timed automata that can be verified using Uppaal. The simple PLCs discussed in this paper use the TON timers discussed in Section 3.1.

In [4], PLCs designed graphically using Sequential Function Charts (SFCs) are converted into models that can be analyzed using model checking techniques. Two approaches are introduced. Both Cadence SMV and Uppaal are applied.

In [38], two examples of analyzing PLC applications with Uppaal are presented. The PLC timing is modelled in detail. Especially the reaction time of the PLC to a signal is observed. In the first example the PLC just reads input and writes output. The second example utilizes a TON timer: the timer is set when the PLC receives input. PLC output is set when the timer's timeout is recognized.

A distributed lift system was re-examined in [41]. The lift system was previously found faulty in [25], but the errors were fixed in an *ad hoc* manner by the system developers. In this paper the developers' solutions are analyzed using Uppaal. The solutions are shown to be incorrect. A different solution is proposed and proven to be correct.

In [28] it is shown how model checking can be used in the designing process of a deadlock free wafer scanner controller with an optimized throughput. Deadlock situations are possible, since the wafers are handled by robots that work independently. In their work a deadlock avoidance policy is first analyzed based on a finite state model using the SMV model checker. A throughput analysis can then be performed on a more detailed timed automaton model using the Uppaal tool. The two models (the SMV model and the Uppaal model) are formally related through the notion of a stuttering bisimulation introduced by Browne et al. [14]. In the throughput optimization part an observer automaton is used to measure the progress of the system. It observes the unload events of wafers that had already been scanned. The observer is used to find an infinite schedule that takes at most $H$ time units until the first unload event, and that has at most $S$ time units between two

unload events.

In [13] a wafer scanner system is analyzed using model-based techniques. As a part of their analysis the system is modelled with the process algebraic language $\chi$ (Chi) [47]. The model is then translated into Uppaal timed automata. Some properties are then verified using the Uppaal model.

In [11] a turntable system is analyzed using various model checking tools. A $\chi$ (Chi) [47] simulation model is translated into model written in the input languages of CADP [22], SPIN and Uppaal. They concluded that Uppaal is the easiest to translate to. Verification of properties with fairness constraints [20] can be done in CADP and SPIN but is impossible in Uppaal.

In [37] a prototype gear controller is designed and analyzed. Uppaal is used to verify the design. The paper also introduces a method to verify bounded response time properties in Uppaal without an observer automaton. Extra variables (boolean variables and clock variables) are used instead.

### 4.3 Other Real-Time Research

In [30], techniques for generating timed test cases are introduced. In the paper the examined environment and the system under test are both modelled as timed automata. Using the presented techniques, both single purpose, and coverage based test cases are then obtained as counterexamples given by Uppaal.

## 5 CASE STUDY: FALCON

### 5.1 The Falcon Arc Protection System

The Falcon arc protection system is designed by the engineering company Urho Tuominen Oy (UTU). The system is designed to increase personnel safety and minimize material damages in case of an electric arc, e.g., in switchgear. This is accomplished by cutting off the electricity, when an electric arc is observed. The Falcon system consists of the UTU-Falcon master unit, several light sensors and several current sensors. The master unit is implemented as a programmable logic controller (PLC). It is possible to design new logical protection architectures that can be uploaded to the master unit. An example of such a logic is in Figure 10.

The devices used to cut electricity are circuit breakers. Circuit breakers are automatically-operated switches that operate in a matter of milliseconds. Their operation is somewhat similar to fuses, but circuit breakers can be reset either automatically or manually.

The master unit operates in cycles. On each cycle, the master unit reads the inputs from its sensors and sends output signals according to its pro-

grammed logic. Typically the master unit reacts to simultaneous light and current alarms by giving a tripping command to the circuit breakers.

The arc protection system can also be designed to work selectively. This means that electricity is not cut off in every part of the protected area. Instead, only the affected areas are cut off. If the electrical arc does not disappear, an even larger area must be shut down. This kind of selective behaviour is created by placement of the breakers, and the delay components inside the Falcon master unit. The delay components are TON timers introduced in Section 3.1. Intuitively, as soon as an alarm is detected, it is responded to. If the alarm does not disappear, the TON timers in the master unit logic receive continuous input. After some preprogrammed number of cycles, the TON timers give out a signal to some secondary breakers. The secondary breakers will cut off the electricity more extensively.

## 5.2 System Environment Description

The environment and architecture used in this case study is identical to the one used in [45]. The architecture of this case study is made by Matti Koskimies. It is used here with his permission. The architecture is represented in Figure 9.

Electricity is distributed by two distinct power sources. The protected area is divided into three different zones. Each zone has a primary breaker (breakers A, B, D), that will cut the zone off the power network, if an arc is detected. The breaker C is used to separate the power sources. The breaker C is always tripped, when an arc is observed. The secondary breaker G is tripped if the alarm has not disappeared after tripping the primary breaker D. The secondary breaker H is tripped if the alarm still has not disappeared after tripping C, D and G. Similarly, the secondary breaker E is tripped if the alarm has not disappeared after tripping the primary breaker A or B (or both). The secondary breaker F is tripped if the alarm still has not disappeared after tripping A (or B), C and E. The secondary breakers are not tripped immediately after the primary breakers. Some time must first pass so that the primary breakers have time to operate.

The Falcon master unit logic that actualizes this behaviour is in Figure 10. The logic consists of seven input signals, AND gates, OR gates, four delay gates, and eight output signals. Four of the input signals (Cr_1, Cr_2, Cr_3a, Cr_3b) are current alarms signals. The remaining three input signals (L_1, L_2, L_3) are the light alarm signals.

The output signals can be divided into two groups. The fast triac outputs (Triac 1, Triac 2, Triac 3, Triac 4) trigger the primary circuit breakers (A, B, C, D). The slower relay outputs (Relay 1, Relay 2, Relay 3, Relay 4) trigger the secondary circuit breakers (E, F, G, H).

The delay gates are the TON timers introduced in Section 3.1. A signal passes through a delay gate when the gate receives a preset amount of succes-

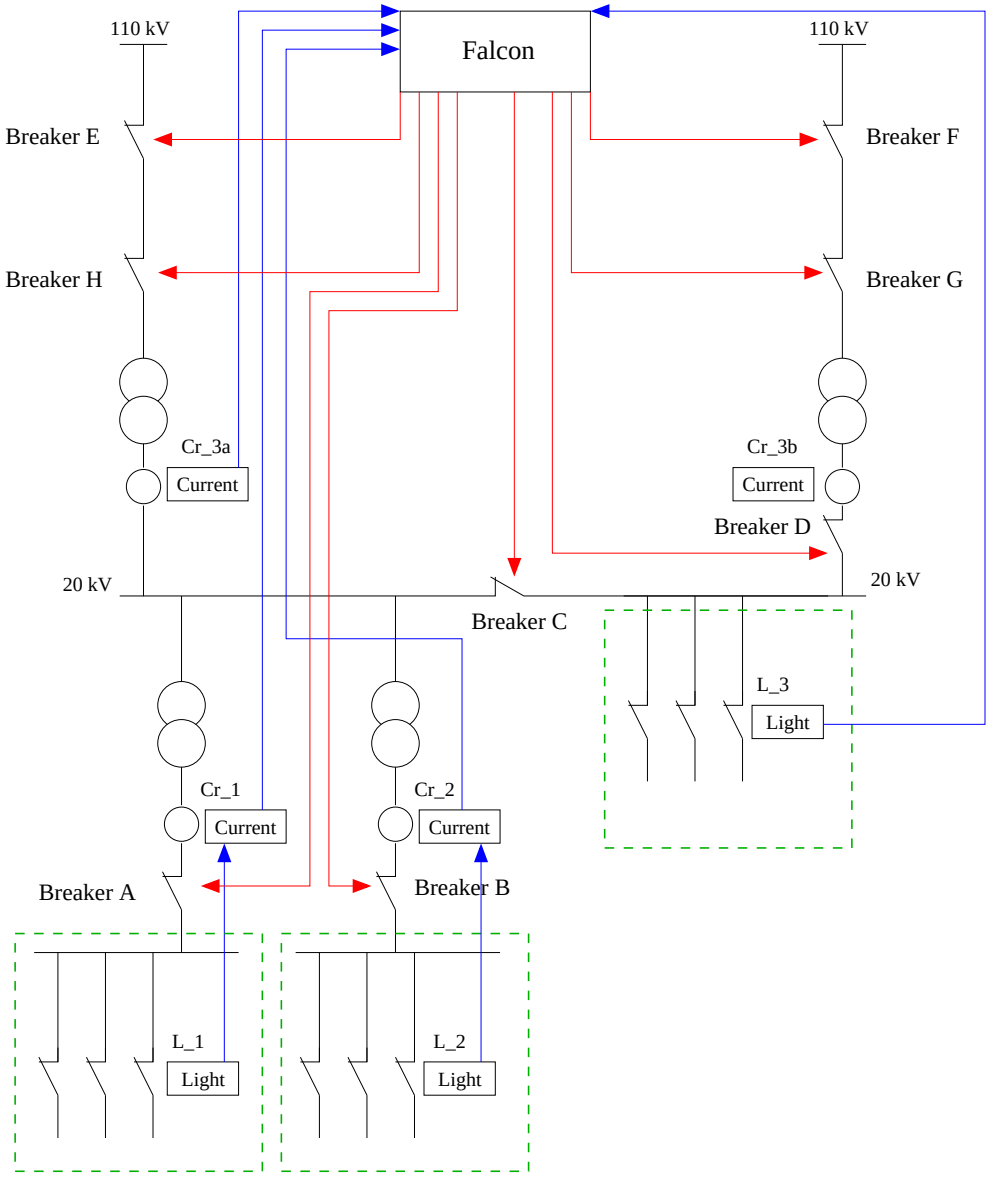


Figure 9: The Falcon architecture

sive input signal alarms. The idea is to let the primary breakers try to solve the problem first. After the preset time limit, the secondary breakers take action.

It is important that an arc protection system does not cut the electricity in vain. A single current alarm, or a single light alarm typically does not indicate an electric arc. Only in case of both of these alarms concurrently from the same area should lead to the cutting of the electricity. On the other hand, the arc protection system should always eventually cut the electricity, if the concurrent current-light alarm does not disappear.

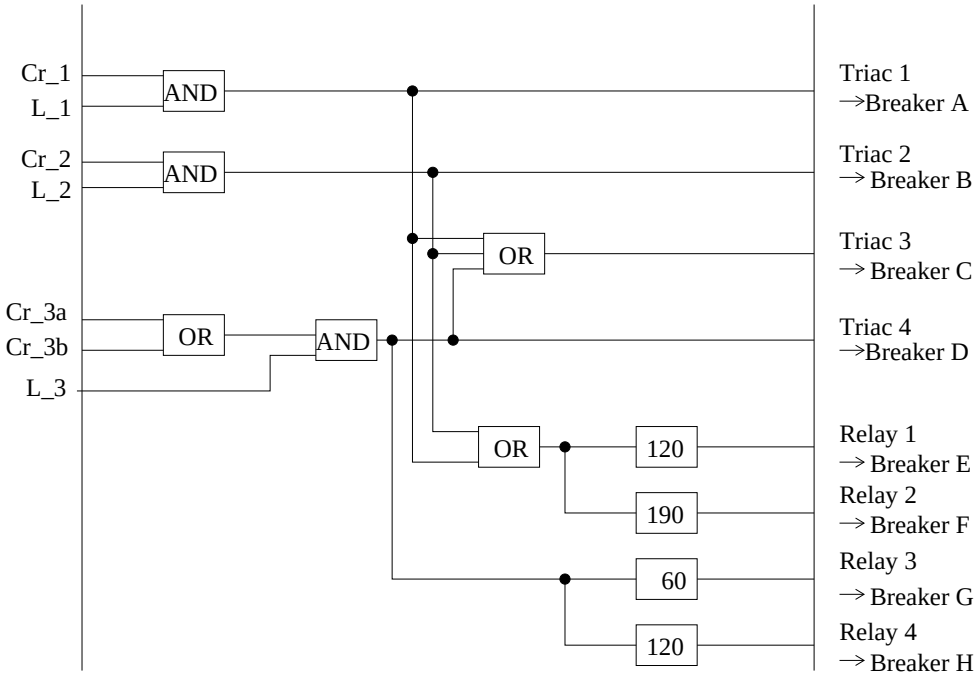


Figure 10: Falcon master unit logic of the example case

## 5.3 Discrete-time Model

A discrete-time model of the Falcon system can be constructed from three different kinds of automata: the Falcon control unit automaton, primary breaker automata, and secondary breaker automata. The primary and secondary breakers need distinct automata since their behaviour in the model is somewhat different. Primary breakers can get broken. However, it is assumed that the secondary breakers are always able to operate. This assumption is made because we are not really interested in situations where all the breakers involved are broken. In these cases it does not really matter how the control unit reacts to the signals.

Next, the model is explained in more detail. The automata are represented as figures. All the code related to the discrete-time model of the Falcon case is in Appendix A. This includes global declarations, template instantiations and the system composition.

### 5.3.1 The Falcon Control Unit

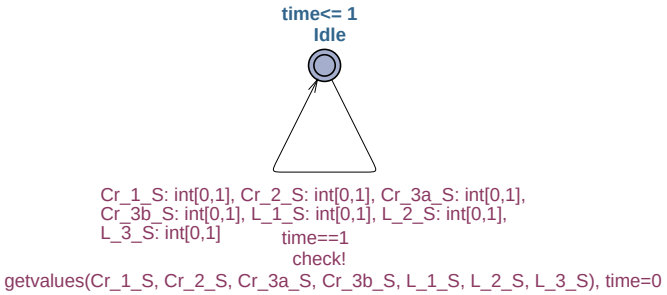


Figure 11: The Falcon system model with discrete time

The timed automaton of the Falcon control unit is in Figure 11. The local declarations of the automaton are in Appendix A. The automaton has only one state, *Idle*. There is only one transition from that state to itself. In addition, the automaton has seven boolean input variables, seven integer variables for internal calculations, and one clock variable. The automaton is constructed so that the only transition is taken repeatedly on constant time intervals, since the real Falcon system operates similarly. This behaviour is forced by an invariant constraint on the clock variable $time$ : $time <= 1$ and a guard constraint $time == 1$ in the transition. The combination of these constraints forces the automaton to take the transition exactly when $time == 1$. The transition is taken repeatedly because the clock variable is also reset to zero during the transition.

So far we have accomplished a transition that is taken on constant intervals. In addition to the guard constraint, the transition has three other parts: selection, updating and synchronization. All the parts are executed at the same time point.

The selection part of the transition ($Cr\_1\_S : int[0, 1], Cr\_2\_S : int[0, 1]$, $Cr\_3a\_S : int[0, 1], Cr\_3b\_S : int[0, 1], L\_1\_S : int[0, 1], L\_2\_S : int[0, 1]$, $L\_3\_S : int[0, 1]$) introduces seven boolean variables that are given a boolean value nondeterministically. These values are later used to determine the values of the overcurrent signal inputs $Cr\_1$, $Cr\_2$, $Cr\_3a$, $Cr\_3b$ and the light signal inputs $L\_1, L\_2, L\_3$. The input values are not given values directly because it might be the case that electricity has already been cut off in a way that some overcurrent signals are impossible. Therefore only intermediate values for the inputs are chosen. The real input values are then filtered from these intermediate values.

The update part of the transition ($getvalues(Cr\_1\_S, Cr\_2\_S, Cr\_3a\_S$, $Cr\_3b\_S, L\_1\_S, L\_2\_S, L\_3\_S), time = 0$) resets the clock and calls the function
*getvalues* with the selected suggestions for inputs as parameters. The function *getvalues* is in Appendix A. The objective of this function is to determine the input values, and calculate the outputs using these inputs. First, it is calculated whether electricity is available in the three different zones of the system. This information can be concluded because we know whether

each breaker has cut or not. Secondly, the sensors can detect an overcurrent only if the sensor is connected to the zone it is observing. There is a breaker C between zone 3 and the sensor detecting the value $Cr\_3a$. The breaker D is between zone 3 and the sensor detecting the value $Cr\_3b$. Using logical AND, the actual input values are calculated.

Using the inputs and the logic chosen for the system, the fast triac outputs are easy to calculate. The delayed outputs are slightly more complicated. Each of these outputs are associated with a TON timer introduced in Section 3.1. Three variables are used for every timer-output pair: $relayN, rN$ and $relNbuffer$, where $N \in \{1, 2, 3, 4\}$ is the number of the output variable. $rN$ is an integer given as a parameter to the Falcon control unit automaton. It determines how many Falcon cycles the boolean variable $relayN$ has to be true until the information is sent to the breakers. $relNbuffer$ calculates the number of consecutive cycles $relayN$ has been true. When the number of cycles is insufficient the variable $relayN$ is reset to zero. This is done because the variable is declared globally. Each secondary breaker observes the value of one of these variables. If $relayN == true$ is detected by them, it indicates that also $relNbuffer == rN$. The behaviour of the function $getvalues$ is atomic. Breakers can not detect the temporary $relayN == true$ values if the variable is reset later in the function.

Finally, the inputs are reset. This is done to avoid state space explosion. If a property that refers to the inputs is checked, it will be necessary to remove the last lines from this function.

The last part of the control unit automaton's transition is the synchronization. It is a way of communication in the system. The message ($check!$) is sent to every breaker. Because of the message, every breaker can react to the new output values instantaneously. Intuitively, the synchronization here means: "Falcon has new outputs, react immediately".

### 5.3.2 Primary Breakers

The timed automaton representing the discrete-time primary breaker is in Figure 12. The automaton has the clock $btime$ declared locally. The automaton has four parameters: the boolean variable that initiates the action (triac), the minimum time needed for cutting the current (mintime), the maximum time needed for cutting the current (maxtime) and the global variable (cuts) that the breaker will set as true, when it has cut the current.

The breaker automaton is intended to take the transition from the state *Checking* to the state *Cutting* when the observed signal becomes true. If the state *Cutting* is reached, the transition from the state *Cutting* to the state *Cut* is inevitably taken between time points $mintime$ and $maxtime$, the threshold values included. The automaton, however, as the observed signal becomes true, can instead choose to take the transition from *Checking* to *Broken*. The state *Broken* is a sink state that models the behaviour of a broken breaker. After the state *Broken* is reached, the automaton can not reach any other state in the automaton.

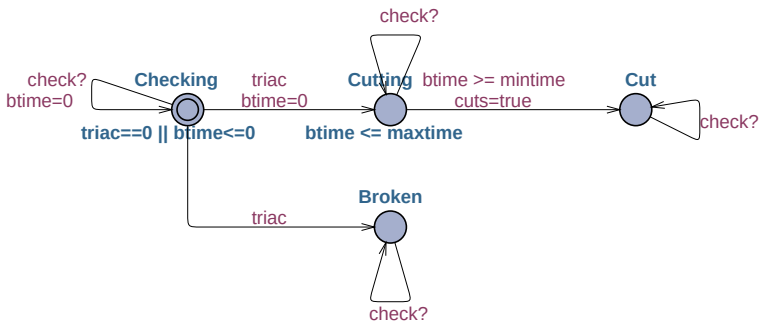


Figure 12: The discrete-time breaker model

The primary breaker automaton initiates the current cutting action immediately after the triac signal is turned on. This is because of the invariant in the *Checking* state $(triac == 0||btime <= 0)$ states that whenever the value of triac is true, the clock variable *btime* must be zero. The transition from *Checking* to itself includes a synchronization and the reset of the clock. This transition is taken together with the Falcon control unit automaton's transition. If the value of the variable triac changes during the Falcon's calculations, the breaker automaton must choose either the transition from *Checking* to *Cutting* or the transition from *Checking* to *Broken*. It can not remain in the state *Checking* and advance in time because the invariant prevents that.

### 5.3.3 Secondary Breakers

The timed automaton representing the discrete-time secondary breaker is in Figure 13. The model has the clock *btime* declared locally. Secondary breakers are almost identical to the primary breakers. The only difference is the absence of the *Broken* state and the transition leading to that state. This means that after observing a triac signal, a secondary breaker automaton will always end up in the state *Cut*. This will happen between time points *mintime* and *maxtime*, the threshold values included.

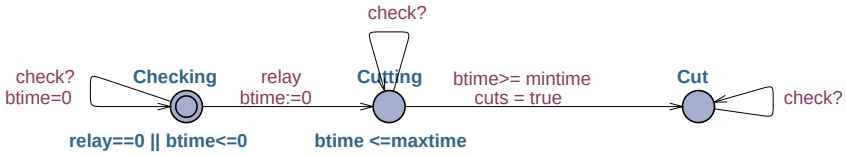


Figure 13: The discrete-time secondary breaker model

## 5.4 Falcon: Continuous-time Model

The fundamental idea of the continuous-time model is the absence of the Falcon's constant interval clock cycle. In fact, there is no clock variables in the Falcon control unit automaton. The Falcon automaton's operation is controlled by a different environment automaton. The environment automa-

ton decides when the Falcon will operate.

Since the Falcon clock cycle is missing, the continuous-time model can experience zenoness, i.e., an infinite number of Falcon operations can occur in finite time. Therefore, some of the liveness properties checked against the discrete-time model can not be verified as such. For these properties, a separate observer automaton has to be created. The properties are transformed into bounded liveness properties that imply the unbounded versions of the same properties.

### 5.4.1 The Falcon Control Unit

The timed automaton representing the continuous-time Falcon system is in Figure 14. The local declarations of the Falcon control unit automaton are in Appendix B.

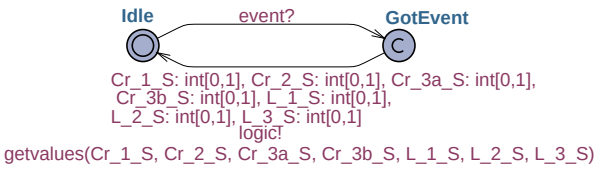


Figure 14: The falcon system of the continuous-time model.

In essence the automaton is very similar to the discrete-time Falcon control unit automaton. The clock variable *time* is not needed, and a second state has been added. The transition from the state *Idle* to the committed state *GotEvent* is taken when an *event* synchronization is received. Because the second state is committed, the transition back to the state *Idle* must follow without any time elapsing. When the transition from the committed state to the state *Idle* is taken, the Falcon system will operate as in the discrete-time case. New inputs are sampled, and the outputs are derived from the inputs.

### 5.4.2 Primary Breakers

The timed automaton of the continuous-time primary breaker is in Figure 15. The automaton is again very similar to the discrete time primary breaker automaton. The only difference is that a new committed state *Decide* has been added. The automaton used in the discrete-time case would not work properly in the continuous-time model. In the discrete-time model the invariant in the state *Checking* stated that the clock *btime* had to be zero or the signal had to be false. In the case of an alarm, some transition had to be taken out of the state *Checking* before time could advance.

In the continuous-time model, however, it would not be necessary for the model to take the transition out of the state *Checking*. It would be possible to sample a new event and not advance in time at all. This way an alarm signal could get unnoticed. The committed state in the continuous-time model is added to force the breaker to react to a possible alarm. If some automaton is in a committed state, only transitions leaving from a committed state are

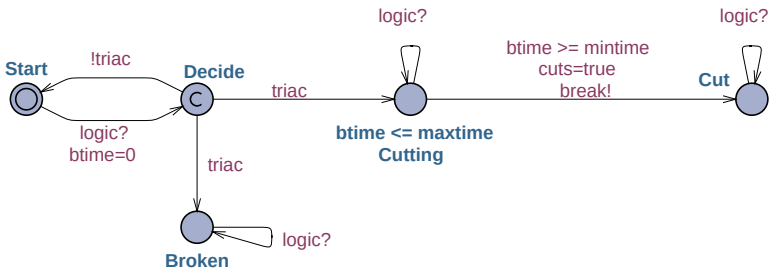


Figure 15: The continuous-time model of the breaker

allowed in all automata.

### 5.4.3 Secondary Breakers

The timed automaton of the continuous-time secondary breaker is in Figure 16. As with the primary breaker automaton, committed states are used to force the breaker to react to an alarm signal.

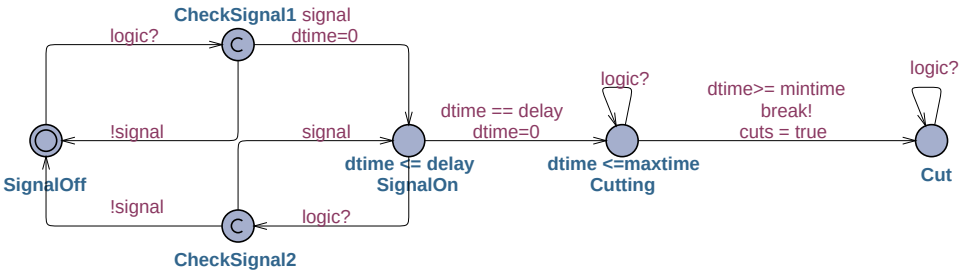


Figure 16: The secondary breaker model in continuous-time

The secondary breaker automaton has new features in the continuous-time model. This is because the Falcon automaton can not use integer counters for the slow secondary breaker signals. It is not demanded that the continuous-time Falcon automaton operates regularly, and the integer counters in the discrete-time case were only incremented during the Falcon cycle. In Appendix B the TON timer behaviour has been removed from the Falcon automaton's code. Substitutive behaviour is added to the respecting secondary breaker automata. A new parameter *delay* has been added for the TON timer implementation.

The purpose of the automaton is to observe its signal, and trip when necessary. The signal value can only change when Falcon operates. Falcon operation is synchronized with transitions to the committed states *CheckSignal1* and *CheckSignal2* that force response to the signal value. When an alarm signal is observed the first time, the local clock *dtime* is reset and the transition to the state *SignalOn* is taken. When the alarm signal is continuously detected, the local clock *dtime* can become as large as the delay parameter. When this happens, the transition from the state *SignalOn* to the state *Cutting* must be taken. The *SignalOn* invariant constraint and the guard on the transition from *SignalOn* to *Cutting* together state that the state *Cutting* will be reached exactly when the alarm signal has been continuously detected

for *delay* time points. This is the TON timer behaviour included in the automaton. The transition from *SignalOn* to *Cutting* models the time instant when the tripping decision is made in the Falcon control unit logic.

When the *Cutting* state is reached, the remaining functionality of the automaton is related to the breaker behaviour. The same clock variable *dtime* is used to model the breaker's tripping time. This variable is reset when the state *Cutting* is reached. Finally, the state *Cut* is reached between time points *mintime* and *maxtime*.

### 5.4.4 The Environment Model

The timed automaton of the continuous-time environment model is in Figure 17.

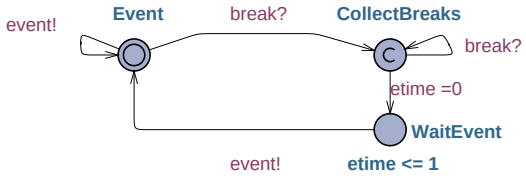


Figure 17: The continuous-time environment model

The environment decides when the Falcon automaton operates (i.e., collects inputs and produces output). The transition from the state *Event* to itself can be taken anytime. The transition is synchronized with the Falcon control unit automaton's transition from the state *Idle* to the state *GotEvent*. The Falcon operation transition immediately succeeds.

Additionally, the environment automaton also has a synchronization with the breakers via the *break* channel. For physical reasons, after a breaker has cut, the current sensors should not be able to sense any current. This is why the sensor values have to be updated accordingly. In the discrete case, this is not a concern since the values are always updated on the next clock cycle. When a breaker cut and a Falcon operation happen exactly at the same time, the order of these events is chosen non-deterministically. If the breaker cut event is executed before the Falcon operation, the disappearance of an alarm is detected properly. If the Falcon operation is executed before the breaker cut event, the effects of the cut can not be noticed until the next Falcon operation.

In continuous time, however, the Falcon control unit does not operate on a specific rate. After a breaker has cut the electricity, a Falcon operation might not be summoned at all and sensor values could remain in an alarming state even though a current alarm from a specific zone was not even possible. Thus, the environment model must be modified so that all the relevant behaviour of the discrete-time model is modelled. The solution here is to modify the environment so that after a breaker cuts, an event always follows. As in the discrete-time case, the cutting of a breaker can be detected immediately or on the next Falcon operation. After a breaker cut, the transition to

the *WaitEvent* state is taken immediately. From this state the transition to *Idle* can be taken immediately. This transition synchronizes with the Falcon automaton. It is also possible to wait in this state for at most one clock cycle, and then synchronize with the Falcon automaton. The Falcon operation transition immediately succeeds.

In the model it is possible for multiple breakers to cut at the same time. In the committed *CollectBreaks* state, it is possible to receive other *break* synchronizations as well. The model also allows the handling of the *break* synchronizations separately, so that a Falcon operation is between every *break* synchronization. In reality, this kind of behaviour does not happen. In some cases two different inputs generated at the same time point could result in unwanted behaviour. Two distinct breakers could be tripped simultaneously even though an output that trips both breakers at the same time were impossible. This is an over-approximation: the model has more behaviour than the actual system. For example, the model can be run at a specific rate, including the actual rate of the modelled system. However, the model can also be run at a much more faster rate. Even infinite operating frequency is possible in the model. If safety properties can be verified with an over-approximated model, they are also valid in the actual system.

## 5.5 Checked Properties

The two models were both verified against five properties. The checked properties were:

- **Property 1:** If the secondary breaker E cuts, breaker A or breaker B is broken. Using Uppaal's TCTL this is:

  ```
  A[] (BreakerE.Cut imply (BreakerA.Broken or
       BreakerB.Broken))
  ```

- **Property 2:** If Breaker A cuts, a current-light alarm has been observed in the past. Using Uppaal's TCTL this is:

  ```
  A[] (breakerAcuts imply was_Cr_1)
  ```

  For this property, the variable $was\_Cr\_1$ has to be added to the model. Also, a line has to be added to the Falcon system -model:

  ```
  if (Cr_1 && L_1) was_Cr_1 = true;
  ```

- **Property 3:** A simultaneous current and light alarm leads to the disappearance of the alarm, or to the cutting of breaker A or breaker E. Using Uppaal's TCTL this is in the discrete-time case:

  ```
  (Falcon.Cr_1 & Falcon.L_1) --> (not Falcon.Cr_1 or
  not Falcon.L_1 or breakerAcuts or breakerEcuts)
  ```

In the continuous-time case the property is:

```
 A[] not(Observer2.AfterTimeBound and not breakerAcuts
and not breakerEcuts and Cr_1 and L_1)
```

The observer automaton in Figure 18 is used. The observer is such that the state *AfterTimeBound* can only be reached when the current signal $Cr\_1$ and the light signal $L\_1$ have been true for 170 time points. The constant is chosen so that the breakers are given enough time to operate.

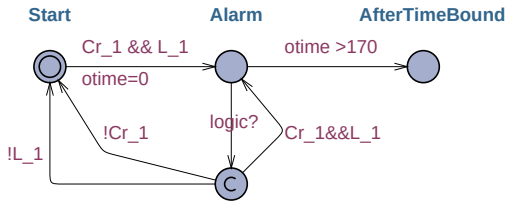


Figure 18: The observer automaton used in property 3 in the continuous-time case

- **Property 4:** Continuous alarm longer than the secondary breakers E and H require leads to the cutting of breaker E or breaker H. Using Uppaal's TCTL this is in the discrete-time case:

```
Obs7.Current_7_cycles --> (breakerEcuts or
                           breakerHcuts)
```

The observer automaton in Figure 19 is used. The variable *Now_CrL* of the automaton must be globally declared and properly updated depending on the current and light signals. *Now_CrL* is updated during each Falcon cycle. If both the current and the light signal are true, then *Now_CrL* is set to *true*. Otherwise *Now_CrL* is set to *false*. In the observer automaton the counter variable *cycles* is updated accordingly: $cycles = (cycles + 1) * Now\_CrL$. That is, the counter is increased by 1 if *Now_CrL* is true. Otherwise the counter is set to 0.

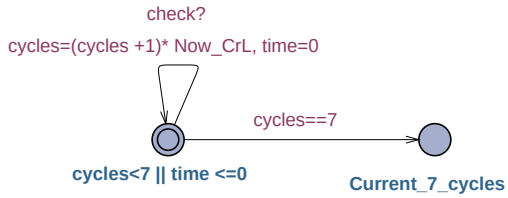


Figure 19: An observer automaton for discrete-time

In the continuous-time case the property is:

```
A[] Observer.AfterLongAlarm imply (breakerEcuts or
                                   breakerHcuts)
```

The observer automaton in Figure 20 is used. The automaton is such that one of the committed states *Detect1* or *Detect2* is reached whenever Falcon operates. When the alarm signal is detected the first time, the state *Alarm* will be reached. If the alarm signal does not disappear in 120 time units, the transition from *Detect2* to the state *LongAlarm* is taken. The constant is chosen so that the breakers are given enough time to operate. From this state the transition to *AfterLongAlarm* is eventually taken.

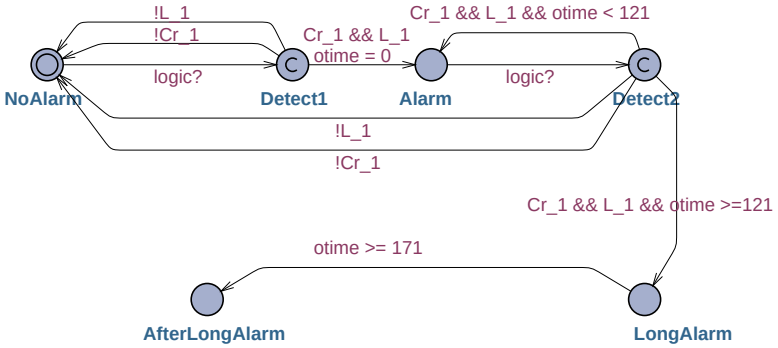


Figure 20: An observer automaton for continuous-time model

- **Property 5:** Continuous alarm longer than the secondary breaker G requires leads to electricity cut from all zones. Using Uppaal's TCTL this is in the discrete-time case:

```
Obs10.Current_10_cycles --> ((breakerEcuts or
          breakerHcuts) and (breakerCcuts or
          breakerFcuts or breakerGcuts or breakerDcuts))
```

In the continuous-time case the property is:

```
A[] Observer.AfterLongAlarm imply ((breakerEcuts or
          breakerHcuts) and(breakerCcuts or breakerFcuts
          or breakerGcuts or breakerDcuts))
```

The observer automaton in Figure 20 is used with timing modifications. The modified version is in Figure 21.

## 5.6 Conclusions of the Models

In the continuous-time model the Falcon operation can happen at any time point. The continuous-time model also ensures that at least one Falcon operation occurs after a breaker has cut the electricity. Thus, the execution traces it produces is a superset of the execution traces of any version of the discrete-time model. The continuous-time model simulates the discrete-time model. The continuous-time model even has more behaviour than the system in real life. It is an over-approximation of the real system. If safety properties can be verified on the continuous-time model, they must also be valid on the real

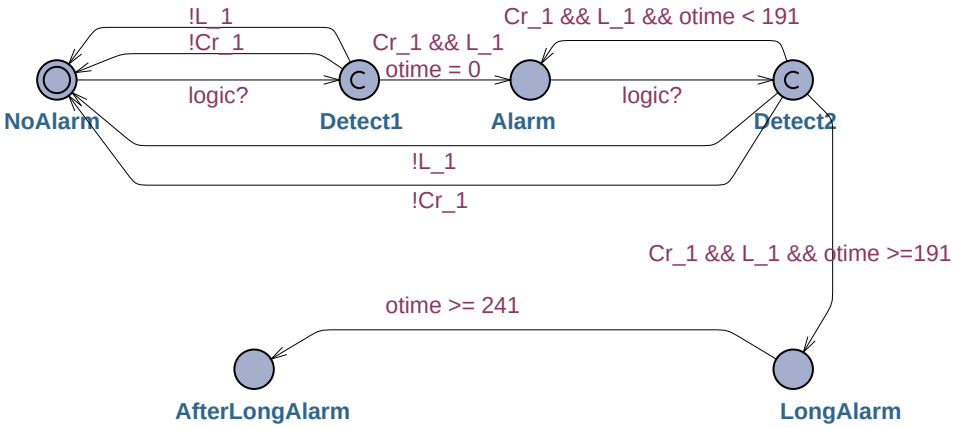


Figure 21: The observer automaton used in Property 5

system.

The properties 3, 4 and 5 are transformed into bounded liveness properties in the continuous-time case. The properties give an explicit time bound for the property, and are thus tighter than the properties in the discrete-time case.

## 6 RESULTS

The five properties were checked against the continuous-time model and different versions of the discrete-time model. The verification was done on a standard PC with 1.8GHz Inter Core 2 Duo E63xx DualCore processor. The available virtual memory was limited to 1.5GiB. Uppaal version 4.0.6 was used in the verification. The discrete-time model versions differed only in operation frequency of the Falcon control unit. In reality, the Falcon control unit operates at least 50 times in the time it takes a breaker to cut the current. In the discrete-time model this value could not be used, since the calculations became too complex. Instead, from 2 to 12 Falcon operations were used in the discrete-time models.

The discrete-time models checked were: discrete_2, discrete_4, discrete_6, discrete_12 and discrete_2-4.

In the model discrete_2, Falcon operates two times in the time it takes a breaker to cut the current. The delays in the Falcon control unit were changed according to the breaker tripping time. For the model discrete_2 the delays of the secondary breakers E, F, G and H were 6, 9, 3 and 6 Falcon cycles. The delay values are chosen according to the system architecture. Secondary breakers can not trip in vain.

In the model discrete_4, Falcon operates four times in the time it takes a breaker to cut the current. Secondary breaker delays were 10, 15, 5 and 10 Falcon cycles.

In the model discrete_6, Falcon operates six times in the time it takes a

breaker to cut the current. Secondary breaker delays were 14, 21, 7 and 14 Falcon cycles.

In the model discrete_12, Falcon operates twelve times in the time it takes a breaker to cut the current. Secondary breaker delays were 26, 39, 13 and 26 Falcon cycles.

The model discrete_2-4 was such that breakers needed a minimum of 2 cycles and a maximum of 4 cycles to trip. Secondary breaker delays were 10, 15, 5 and 10 Falcon cycles.

The continuous-time model was such that breakers needed 50 time units to trip. Secondary breaker delays were 120, 190, 60 and 120 time units.

The running times were as follows:

| | discrete_2 | discrete_4 | discrete_6 | discrete_12 |
|---|---|---|---|---|
| Property 1 | 1 min 11 s | 8 min 29 s | 32 min 22 s | 16 h 27 min |
| Property 2 | 1 min 11 s | 8 min 31 s | 38 min 37 s | 16 h 28 min |
| Property 3 | 1 min 19 s | 9 min 30 s | 43 min 19 s | 19 h 10 min |
| Property 4 | 2 min 19 s | 19 min | 1 h 38 min | $>$ 20 h |
| Property 5 | 2 min 7 s | 17 min | 1 h 27 min | Out of Memory |

| | discrete_2-4 |
|---|---|
| Property 1 | 7 min 28 s |
| Property 2 | 7 min 25 s |
| Property 3 | 9 min 25 s |
| Property 4 | 19 min 13 s |
| Property 5 | 17 min 18 s |

As a comparison, if a simplification to the Falcon logic is made in the discrete-time model, so that no light signals exist, but the same functional behaviour is present, the following results were obtained. The simplified discrete-time model is represented in Appedix C.

| | abs_discrete_2 | abs_discrete_4 | abs_discrete_6 | abs_discrete_12 |
|---|---|---|---|---|
| Property 1 | 5 s | 34 s | 2 min 44 s | 1 h 54 min |
| Property 2 | 5 s | 34 s | 2 min 43 s | 1 h 53 min |
| Property 3 | 5 s | 39 s | 3 min 6 s | 2 h 14 min |
| Property 4 | 10 s | 1 min 22 s | 7 min 19 s | Out of Memory |
| Property 5 | 10 s | 1 min 23 s | 7 min 47 s | Out of Memory |

| | abs_discrete_2-4 |
|---|---|
| Property 1 | 30 s |
| Property 2 | 30 s |
| Property 3 | 41 s |
| Property 4 | 1 min 22 s |
| Property 5 | 1 min 24 s |

Results for the continuous-time case were the following:

| | continuous-model |
|---|---|
| Property 1 | 30 min 18 s |
| Property 2 | 29 min 56 s |
| Property 3 | 1 h 6 min |
| Property 4 | 2 h 40 min |
| Property 5 | 2 h 39 min |

As in the discrete-time case, the same logic simplification can also be made in the continuous-time case. The simplified continuous-time model is represented in Appedix D. The following results were obtained:

| | abs_continuous -model |
|---|---|
| Property 1 | 1 min 33 s |
| Property 2 | 1 min 32 s |
| Property 3 | 4 min 30 s |
| Property 4 | 16 min 12 s |
| Property 5 | 16 min 15 s |

We can make many observations from the results. Firstly, the checked property itself has influence on the length of the verification in all models. Properties 1 and 2 seem to be the easiest, and properties 4 and 5 seem to be the hardest. The properties that require an observer automaton take invariably more time to verify than the other properties. This is understandable because an additional automaton adds some complexity to the model.

Secondly, the frequency of the Falcon control unit operation in the discrete-time model had a great influence on the verification times. In the model discrete_12 the properties 4 and 5 could not be verified within the used time and memory limit, while in the model discrete_2 all properties could be verified within a few minutes.

If the logic is simplified so that the same functional behaviour is preserved, while the number of inputs is decreased, the verification times decrease substantially in both models. The properties 4 and 5 could still not be verified for the discrete-time model discrete_12.

Also, the use of variable length cutting times in the model discrete_2-4 did not result in longer verification times for any property compared to the model discrete_4.

Finally, while the continuous-time model could not be verified in a particularly short time, it covers all the behaviour of a discrete-time model operating at a real-life frequency. Such a discrete-time model could not be verified because of its complexity.

# 7 CONCLUSIONS

In this work an electric arc protection system controlled by a programmable logic controller (PLC) was modelled and verified. The used model checking tool was Uppaal. The UTU electric arc protection system was modelled in two different ways. The discrete-time model was based on counters and a controller operating at a specific rate. In the continuous-time model the counters were replaced with clocks, and the controller was allowed to operate at a varying frequency.

The properties requiring an observer automaton could not be checked against the version of the discrete-time model with the most complexity, discrete_12. Either the memory limit of 1.5 GiB was reached or more than 20 hours was needed for the verification. In addition, the checked discrete-time model versions were simpler than the real modelled system. To be exact, the real controller operates more frequently than in the discrete-time model. It is also hard to prove that the discrete-time model captures all the essential behaviour of the system.

The discrete-time model was not really the optimal model to be analyzed with the real-time model checker Uppaal because of the discrete time. Uppaal is designed to work with real-time models, and this is why the continuous-time model proved to be more expressive.

The continuous-time model actually has more behaviour than the real controller. Therefore, if no unwanted behaviour could be observed in the continuous-time model, the real system must be working correctly as well. This is only true because the checked properties could be stated as invariant properties i.e., properties that are true in all states of the model.

It can also be noticed that the continuous-time model simulates the discrete-time model operating at a realistic frequency. The set of execution traces of the continuous-time model covers the execution traces of the discrete-time model. In other words, the set of execution traces of the continuous-time model is a superset of the execution traces of the discrete-time model. The continuous-time model can always choose to imitate the operation of the discrete-time model.

As a general observation, the most difficult part of the modelling work was to find the right abstraction level that captures the system behaviour adequately, and does not result in too complex verification. In Uppaal, the running times of the verification increase rapidly as the number of input variables is increased. In the results, the logic simplification was done by removing 3 of the 7 input variables of the Falcon control unit. This simplification led to significant decrease in running times. The external behaviour of the system did not change because of the simplification.

All TCTL specifications can not be stated in the subset of TCTL Uppaal supports. Many specifications require the use of an additional observer au-

tomaton. The use and generation of these observer automata is not straightforward, and the correctness of the observer automaton implementation is difficult to assure.

In conclusion, Uppaal is suitable for the model checking of time-critical systems, but does not perform well if a lot of unconstrained input variables are needed. Model checking is a valuable tool in the verification of safety instrumented systems.


## Acknowledgements

This work was part of the research project Model-based safety evaluation of automation systems (MODSAFE). The project is part of the Finnish Research Programme on Nuclear Power Plant Safety 2007-2010 (SAFIR2010), funded by the State Nuclear Waste Management Fund (VYR).

I want to thank my supervisor Prof. Ilkka Niemelä and my instructor Docent Keijo Heljanko for their encouragement and guidance throughout this study. I also want to thank for the opportunity to work here at the Department of Information and Computer Science.


## REFERENCES


[1] R. Alur. Timed automata. In *Computer Aided Verification*, pages 8–22, 1999.

[2] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real-time systems. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, Pennsylvania, June 4–7 1990. IEEE Computer Society Press.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[4] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg. Verification of PLC programs given as sequential function charts. In *SoftSpez Final Report*, pages 517–540, 2004.

[5] G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi. Uppaal implementation secrets. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 3–22, London, UK, 2002. Springer-Verlag.

[6] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Meth-*

*ods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer-Verlag, September 2004.

[7] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T. A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, pages 244–256, New Brunswick, NJ, USA, / 1996. Springer-Verlag.

[8] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer-Verlag, 2003.

[9] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[10] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5:5), 2006. (doi: 10.2168/LMCS-2(5:5)2006).

[11] E. M. Bortnik, N. Trcka, A. Wijs, B. Luttik, J. M. van de Mortel-Fronczak, J. C. M. Baeten, W. Fokkink, and J. E. Rooda. Analyzing a $\chi$ model of a turntable system using Spin, CADP and Uppaal. *J. Log. Algebr. Program.*, 65(2):51–104, 2005.

[12] B. Brard, P. Bouyer, and A. Petit. Analysing the PGM Protocol with UPPAAL. In *Proc. 2nd Workshop on Real-Time Tools (RT-TOOLS'02)*. Technical Report 2002-025, Uppsala University, Sweden, 2002.

[13] N. C. W. M. Braspenning, E. M. Bortnik, J. M. van de Mortel-Fronczak, and J. E. Rooda. Model-based system analysis using Chi and Uppaal: An industrial case study. *Comput. Ind.*, 59(1):41–54, 2008.

[14] M. C. Browne, E. M. Clarke, and O. Grumberg. Characterizing Kripke structures in temporal logic. In *The International Joint Conference on theory and practice of software development on TAPSOFT '87*, pages 256–270, London, UK, 1987. Springer-Verlag.

[15] K. Cerans. Decidability of bisimulation equivalences for parallel timer processes. In G. von Bochmann and D. K. Probst, editors, *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings*, volume 663 of *Lecture Notes in Computer Science*, pages 302–315. Springer-Verlag, 1992.

[16] Z. Chaochen. Duration calculus, a logical approach to real-time systems. In A. M. Haeberer, editor, *Algebraic Methodology and Software Technology, 7th International Conference, AMAST '98, Amazonia, Brasil, January 4-8, 1999, Proceedings*, volume 1548 of *Lecture Notes in Computer Science*, pages 1–7. Springer-Verlag, 1998.

[17] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. *Lecture Notes in Computer Science*, 2000:176–194, 2001.

[18] P. R. D'Argenio, J.-P. Katoen, T. C. Ruys, and G. J. Tretmans. The bounded retransmission protocol must be on time! In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 416–432, Enschede, The Netherlands, 1997. Springer-Verlag, Lecture Notes in Computer Science 1217.

[19] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[20] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[21] E. A. Emerson. Temporal and modal logic. pages 995–1072, The MIT Press, 1990.

[22] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In R. Alur and T. A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 437–440, New Brunswick, NJ, USA, / 1996. Springer-Verlag.

[23] W. M. Goble and H. Cheddie. *Safety Instrumented Systems Verification: Practical Probabilistic Calculations*. ISA-Instrumentation, Systems, and Automation Society, NC, USA, 2005.

[24] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[25] J. F. Groote, J. Pang, and A. G. Wouters. Analysis of a distributed system for lifting trucks. Technical Report UMI Order Number: SEN-R0111, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 2001.

[26] K. Havelund, K. G. Larsen, and A. Skou. Formal verification of a power controller using the real-time model checker UPPAAL. *Lecture Notes in Computer Science*, 1601:277–298, 1999.

[27] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using uppaal. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 2–13, Washington, DC, USA, 1997. IEEE Computer Society.

[28] M. Hendriks, B. van den Nieuwelaar, and F. Vaandrager. Model checker aided design of a controller for a wafer scanner. *Int. J. Softw. Tools Technol. Transf.*, 8(6):633–647, 2006.

[29] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

[30] A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-optimal real-time test case generation using Uppaal. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2003.

[31] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[32] R. Huuck. *Software Verification for Programmable Logic Controllers*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Germany, 2003.

[33] H. E. Jensen, K. G. Larsen, and A. Skou. Modelling and analysis of a collision avoidance protocol using SPIN and UPPAAL. In *The Second Workshop on the SPIN Verification System, volume 32 of DIMACS, Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[34] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[35] K. G. Larsen and Y. Wang. Time-abstracted bisimulation: Implicit specifications and decidability. *Information and Computation*, 134(2):75–101, 1997.

[36] M. Laursen, R. G. Madsen, and S. K. Mortensen. Verifying distributed LEGO RCX programs using UPPAAL. Technical report, Institute of Computer Science, Aalborg University, 1999.

[37] M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear controller. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 281–297, London, UK, 1998. Springer-Verlag.

[38] A. Mader. Precise timing analysis of PLC applications two small examples. Unpublished manuscript. Available from `citeseer.ist.psu.edu/mader00precise.html`.

[39] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[40] X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: theory and application. *Inf. Comput.*, 114(1):131–178, 1994.

[41] J. Pang, B. Karstens, and W. Fokkink. Analyzing the redesign of a distributed lift system in UPPAAL. In J. S. Dong and J. Woodcock, editors, *ICFEM*, volume 2885 of *Lecture Notes in Computer Science*, pages 504–522. Springer-Verlag, 2003.

[42] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58(1-3):249–261, 1988.

[43] G. Rodriguez-Navas, J. Proenza, and H. Hansson. Using UPPAAL to model and verify a clock synchronization protocol for the controller area network. *ETFA 2005. 10th IEEE Conference on Emerging Technologies and Factory Automation*, 2, 2005.

[44] G. Rodriguez-Navas, J. Proenza, and H. Hansson. An UPPAAL model for formal verification of master/slave clock synchronization over the controller area network. *Factory Communication Systems, 2006 IEEE International Workshop on*, pages 3–12, June 27, 2006.

[45] J. Valkonen, V. Pettersson, K. Björkman, J.-E. Holmberg, M. Koskimies, K. Heljanko, and I. Niemelä. Model-based analysis of an arc protection and an emergency cooling system. VTT Working Papers 93, VTT Technical Research Centre of Finland, Espoo, 2008.

[46] A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1996.

[47] D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Syntax and consistent equation semantics of hybrid Chi. *J. Log. Algebr. Program.*, 68(1-2):129–210, 2006.

[48] F. Wang. Red: Model-checker for timed automata with clock-restriction diagram. In P. Pettersson and S. Yovine, editors, *Workshop on Real-Time Tools, Aalborg University Denmark*. number 2001-014 in Technical Report. Uppsala University, 2001.

[49] H. X. Willems. Compact timed automata for PLC programs. Technical Report CSI-R9925, University of Nijmegen, Computing Science Institute, 1999.

[50] W. Yi. CCS + Time = An interleaving model for real time systems. In *Proceedings of the 18th international colloquium on Automata, languages and programming*, pages 217–228, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[51] S. Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.

# A FALCON CASE: DISCRETE-TIME MODEL RELATED CODE

The **global declarations** used in the model checking of the falcon case in discrete time are listed below.

```
bool tr1, tr2, tr3, tr4;
bool relay1, relay2, relay3, relay4;
broadcast chan check;

bool breakerAcuts = false;
bool breakerBcuts = false;
bool breakerCcuts = false;
bool breakerDcuts = false;
bool breakerEcuts = false;
bool breakerFcuts = false;
bool breakerGcuts = false;
bool breakerHcuts = false;
```

The **system declarations** (parallel composition descriptions) of the case are the following:

```
//Parameters: Secondary breaker delays
Falcon = Falconsystem(6, 9, 3, 6);

//Parameters:
//  1. the triac/relay signal
//  2. minimum cutting time
//  3. maximum cutting time
//  4. boolean variable Breaker-has-cut
BreakerA = Breaker(tr1, 2, 2, breakerAcuts);
BreakerB = Breaker(tr2, 2, 2, breakerBcuts);
BreakerC = Breaker(tr3, 2, 2, breakerCcuts);
BreakerD = Breaker(tr4, 2, 2, breakerDcuts);

BreakerE = BreakerSec(relay1, 2, 2, breakerEcuts);
BreakerF = BreakerSec(relay2, 2, 2, breakerFcuts);
BreakerG = BreakerSec(relay3, 2, 2, breakerGcuts);
BreakerH = BreakerSec(relay4, 2, 2, breakerHcuts);

// List one or more processes to be composed into a system.
system Falcon, BreakerA, BreakerB, BreakerC, BreakerD,
       BreakerE, BreakerF, BreakerG, BreakerH;
```

The **Falcon control unit automaton** related code is the following:

```
//input signals
bool Cr_1, Cr_2, Cr_3a, Cr_3b, L_1, L_2, L_3;
```

```
int rel1buffer = 0;
int rel2buffer = 0;
int rel3buffer = 0;
int rel4buffer = 0;

int zone1getcurrent, zone2getcurrent, zone3getcurrent;
clock time;


//Get Falcon system inputs from suggested inputs and
//calculate outputs
void getvalues(bool Cr_1_S, bool Cr_2_S, bool Cr_3a_S,
     bool Cr_3b_S, bool L_1_S, bool L_2_S, bool L_3_S ) {

        zone1getcurrent = !(breakerAcuts || ((breakerEcuts ||
              breakerHcuts) & (breakerCcuts|| breakerDcuts ||
              breakerGcuts || breakerFcuts)));
        zone2getcurrent = !(breakerBcuts || ((breakerEcuts ||
              breakerHcuts) & (breakerCcuts|| breakerDcuts ||
              breakerGcuts || breakerFcuts)));
        zone3getcurrent = !((breakerCcuts || breakerEcuts ||
              breakerHcuts) & (breakerDcuts || breakerFcuts ||
              breakerGcuts));


        Cr_1 = Cr_1_S & zone1getcurrent;
        Cr_2 = Cr_2_S & zone2getcurrent;
        Cr_3a = Cr_3a_S & !breakerCcuts & zone3getcurrent;
        Cr_3b = Cr_3b_S & !breakerDcuts & zone3getcurrent;

        L_1 = L_1_S;
        L_2 = L_2_S;
        L_3 = L_3_S;

// get the outputs

        tr1 = Cr_1 & L_1;
        tr2 = Cr_2 & L_2;
        tr3 = (Cr_1 & L_1) || (Cr_2 & L_2) ||
              ((Cr_3a || Cr_3b) & L_3);
        tr4 = ((Cr_3a || Cr_3b) & L_3);


        relay1 = (Cr_1 & L_1) || (Cr_2 & L_2) ;
        relay2 = (Cr_1 & L_1) || (Cr_2 & L_2) ;
        relay3 = ((Cr_3a || Cr_3b) & L_3);
        relay4 = ((Cr_3a || Cr_3b) & L_3);
```

```
// calculate continuous alarm lengths (relXbuffer) and
// update slow relay-output values (relayX)
        if (relay1 == 0) {rel1buffer =0;}

        if (rel1buffer < r1 & relay1 == 1) {
                rel1buffer++;
                relay1=0;
        }

        if (rel1buffer == r1 && relay1 == 1){
                relay1 = 1;
        }

        if (relay2 == 0) {rel2buffer =0;}

        if (rel2buffer < r2 & relay2 == 1) {
                rel2buffer++;
                relay2=0;
        }

        if (rel2buffer == r2 && relay2 == 1){
                relay2 = 1;
        }

        if (relay3 == 0) {rel3buffer =0;}

        if (rel3buffer < r3 & relay3 == 1) {
                rel3buffer++;
                relay3=0;
        }


        if (rel3buffer == r3 && relay3 == 1){
                relay3 = 1;
        }

        if (relay4 == 0) {rel4buffer =0;}

        if (rel4buffer < r4 & relay4 == 1) {
                rel4buffer++;
                relay4=0;
        }

        if (rel4buffer == r4 && relay4 == 1){
                relay4 = 1;
        }
```

```
//reset the inputs
        Cr_1=0;
        Cr_2=0;
        Cr_3a=0;
        Cr_3b=0;
        L_1=0;
        L_2=0;
        L_3=0;
}
```

## B FALCON CASE: CONTINUOUS-TIME MODEL RELATED CODE

The **global declarations** used in the model checking of the falcon case in continuous time are listed below.

```
// Place global declarations here.

bool tr1, tr2, tr3, tr4;
bool relay1, relay2, relay3,relay4;
broadcast chan logic;
broadcast chan break;
chan event;

bool breakerAcuts = false;
bool breakerBcuts = false;
bool breakerCcuts = false;
bool breakerDcuts = false;
bool breakerEcuts = false;
bool breakerFcuts = false;
bool breakerGcuts = false;
bool breakerHcuts = false;

bool Cr_1, Cr_2, Cr_3a, Cr_3b, L_1, L_2, L_3;
```

The **system declarations** (parallel composition descriptions) of the case are the following:

```
Falcon = Falconsystem();

BreakerA = Breaker(tr1, 50, 50, breakerAcuts);
BreakerB = Breaker(tr2, 50, 50, breakerBcuts);
BreakerC = Breaker(tr3, 50, 50, breakerCcuts);
BreakerD = Breaker(tr4, 50, 50, breakerDcuts);

// parameters:
//      delay-time
//      signal that needs to stay on for the time of delay
//      mintime of breaker cut
//      maxtime of breaker cut
//      the variable that is set when break

BreakerE = DelayAndBreaker(102, relay1, 40, 50, breakerEcuts);
BreakerF = DelayAndBreaker(153, relay2, 40, 50, breakerFcuts);
BreakerG = DelayAndBreaker(51, relay3, 40, 50, breakerGcuts);
BreakerH = DelayAndBreaker(102, relay4, 40, 50, breakerHcuts);


// List one or more processes to be composed into a system.
system Falcon, BreakerA, BreakerB, BreakerC, BreakerD, BreakerE,
```

```
 BreakerF, BreakerG, BreakerH, Environment, Observer;
```

The **Falcon control unit automaton** related code is the following:

```
int zone1getcurrent, zone2getcurrent, zone3getcurrent;


//Get Falcon system inputs from suggested inputs and
//calculate outputs
void getvalues(bool Cr_1_S, bool Cr_2_S, bool Cr_3a_S,
     bool Cr_3b_S, bool L_1_S, bool L_2_S, bool L_3_S) {


//calculate possible input values
        zone1getcurrent = !(breakerAcuts || ((breakerEcuts ||
              breakerHcuts) & (breakerCcuts|| breakerDcuts ||
              breakerGcuts || breakerFcuts)));
        zone2getcurrent = !(breakerBcuts || ((breakerEcuts ||
              breakerHcuts) & (breakerCcuts|| breakerDcuts ||
              breakerGcuts || breakerFcuts)));
        zone3getcurrent = !((breakerCcuts || breakerEcuts ||
              breakerHcuts) & (breakerDcuts ||
              breakerFcuts || breakerGcuts));


        Cr_1 = Cr_1_S & zone1getcurrent;
        Cr_2 = Cr_2_S & zone2getcurrent;
        Cr_3a = Cr_3a_S  & zone3getcurrent & !breakerCcuts;
        Cr_3b = Cr_3b_S  & zone3getcurrent & !breakerDcuts;

        L_1= L_1_S;
        L_2=L_2_S;
        L_3=L_3_S;


// get the outputs
        tr1 = Cr_1 & L_1;
        tr2 = Cr_2 & L_2;
        tr3 = (Cr_1 & L_1) || (Cr_2 & L_2) ||
              ((Cr_3a || Cr_3b) & L_3);
        tr4 = ((Cr_3a || Cr_3b) & L_3);

        relay1 = (Cr_1 & L_1) || (Cr_2 & L_2) ;
        relay2 = (Cr_1 & L_1) || (Cr_2 & L_2) ;
        relay3 = ((Cr_3a || Cr_3b) & L_3);
        relay4 = ((Cr_3a || Cr_3b) & L_3);
}
```

## C FALCON CASE: THE DISCRETE-TIME SIMPLIFIED MODEL

Here the discrete-time model with a logic simplification is represented. Only the modified parts of the model are shown. Otherwise the model remains unchanged. The Falcon control unit automaton of the simplified discrete-time model is in Figure 22.

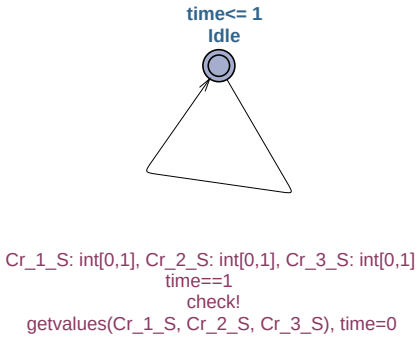


Figure 22: The Falcon control unit of the simplified discrete-time model

The **global declarations** used in the model checking are listed below.

```
bool tr1, tr2, tr3, tr4;
bool relay1, relay2, relay3, relay4;
broadcast chan check;

bool breakerAcuts = false;
bool breakerBcuts = false;
bool breakerCcuts = false;
bool breakerDcuts = false;
bool breakerEcuts = false;
bool breakerFcuts = false;
bool breakerGcuts = false;
bool breakerHcuts = false;

bool was_Cr_1, Now_CrL;
```

The **Falcon control unit automaton** related code is the following:

```
// observer variables
bool Cr1_L1_was;
bool wasalarm;


//Real input values
bool Cr_1, Cr_2, Cr_3;

int rel1buffer = 0;
int rel2buffer = 0;
```

```
int rel3buffer = 0;
int rel4buffer = 0;

int zone1getcurrent, zone2getcurrent, zone3getcurrent;

clock time;

//Get Falcon system inputs from suggested inputs and
//calculate outputs
void getvalues(bool Cr_1_S, bool Cr_2_S, bool Cr_3_S) {


zone1getcurrent = !(breakerAcuts || ((breakerEcuts
     || breakerHcuts) & (breakerCcuts|| breakerDcuts
     || breakerGcuts || breakerFcuts)));
zone2getcurrent = !(breakerBcuts || ((breakerEcuts
     || breakerHcuts) & (breakerCcuts|| breakerDcuts
     || breakerGcuts || breakerFcuts)));
zone3getcurrent = !((breakerCcuts || breakerEcuts ||
     breakerHcuts) & (breakerDcuts || breakerFcuts ||
     breakerGcuts));

Cr_1 = Cr_1_S & zone1getcurrent;
Cr_2 = Cr_2_S & zone2getcurrent;
Cr_3 = Cr_3_S  & zone3getcurrent &
               (!breakerCcuts || !breakerDcuts);

if (Cr_1) was_Cr_1 = true;
if (Cr_1) Now_CrL = true;
if (!Cr_1) Now_CrL = false;


// get the outputs
tr1 = Cr_1;
tr2 = Cr_2;
tr3 = Cr_1 || Cr_2  || Cr_3;
tr4 = Cr_3;

relay1 = tr1 || tr2 ;
relay2 = relay1;
relay3= tr4;
relay4 = tr4;

if (relay1 == 0) {rel1buffer =0;}
if (rel1buffer < r1 & relay1 == 1) {
rel1buffer++;
relay1 =0;
}
if (rel1buffer == r1 && relay1 == 1){
```

```
relay1 = 1;
}

if (relay2 == 0) {rel2buffer =0;}
if (rel2buffer < r2 & relay2 == 1) {
rel2buffer++;
relay2 =0;
}
if (rel2buffer == r2 && relay2 == 1){
relay2 = 1;
}

if (relay3 == 0) {rel3buffer =0;}
if (rel3buffer < r3 & relay3 == 1) {
rel3buffer++;
relay3 =0;
}
if (rel3buffer == r3 && relay3 == 1){
relay3 = 1;
}

if (relay4 == 0) {rel4buffer =0;}
if (rel4buffer < r4 & relay4 == 1) {
rel4buffer++;
relay4 =0;
}
if (rel4buffer == r4 && relay4 == 1){
relay4 = 1;
}

// reset the inputs:

Cr_1=0;
Cr_2=0;
Cr_3=0;
}
```

## D FALCON CASE: THE CONTINUOUS-TIME SIMPLIFIED MODEL

Here the continuous-time model with a logic simplification is represented. Only the modified parts of the model are shown. Otherwise the model remains unchanged. The Falcon control unit automaton of the simplified continuous-time model is in Figure 23.

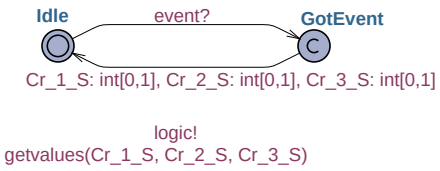


Figure 23: The Falcon control unit of the simplified continuous-time model

The **global declarations** used in the model checking are listed below.

```
bool tr1, tr2, tr3, tr4;
bool relay1, relay2, relay3,relay4;
broadcast chan logic;
broadcast chan break;
chan event;

bool breakerAcuts = false;
bool breakerBcuts = false;
bool breakerCcuts = false;
bool breakerDcuts = false;
bool breakerEcuts = false;
bool breakerFcuts = false;
bool breakerGcuts = false;
bool breakerHcuts = false;

bool Cr_1, Cr_2, Cr_3;
bool was_Cr_1;
```

The **Falcon control unit automaton** related code is the following:

```
int zone1getcurrent, zone2getcurrent, zone3getcurrent;

//Get Falcon system inputs from suggested inputs and
//calculate outputs
void getvalues(bool Cr_1_S, bool Cr_2_S, bool Cr_3_S) {

//calculate possible input values
zone1getcurrent = !(breakerAcuts || ((breakerEcuts
     || breakerHcuts) & (breakerCcuts|| breakerDcuts
     || breakerGcuts || breakerFcuts)));
zone2getcurrent = !(breakerBcuts || ((breakerEcuts
```

```
     || breakerHcuts) & (breakerCcuts|| breakerDcuts
     || breakerGcuts || breakerFcuts)));
zone3getcurrent = !((breakerCcuts || breakerEcuts
     || breakerHcuts) & (breakerDcuts || breakerFcuts
     || breakerGcuts));

Cr_1 = Cr_1_S & zone1getcurrent;
Cr_2 = Cr_2_S & zone2getcurrent;
Cr_3 = Cr_3_S  & zone3getcurrent &
               (!breakerCcuts || !breakerDcuts);


if (Cr_1) was_Cr_1 = true;

// get the outputs
tr1 = Cr_1;
tr2 = Cr_2;
tr3 = Cr_1 || Cr_2  || Cr_3;
tr4 = Cr_3;

relay1 = tr1 || tr2 ;
relay2 = relay1;
relay3 = tr4;
relay4 = tr4;
}
```

TKK REPORTS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-R1 Nikolaj Tatti, Hannes Heikinheimo
Decomposable Families of Itemsets. May 2008.

TKK-ICS-R2 Ville Viitaniemi, Jorma Laaksonen
Evaluation of Techniques for Image Classification, Object Detection and Object Segmentation. June 2008.

ISBN 978-951-22-9444-2 (Print)
ISBN 978-951-22-9445-9 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)