Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor Thesis

From Uppaal To Slab

submitted by
Andreas Abel

submitted
2009

Supervisor
Advisor
Reviewers

Contents

1 Introduction

1.1 Motivation

The increasing prevalence of computer systems in safety-critical areas like air traffic control, nuclear power plants or medical devices entails the need for reliable means to prove their correctness.
A widely used approach to achieve this goal is model checking. It permits to test automatically whether a given mathematical model of a system meets a certain specification. Several such models have been devised and used in a number of model checking tools.
A fairly popular model checking tool is Uppaal, which has been developed at Uppsala University in Sweden and Aalborg University in Denmark. Its modeling language is based on networks of timed automata, which are essentially finite automata extended with clocks. Additionally, it offers features like integer variables, channels or templates. To create a model, the tool features an intuitive graphical user interface.
A different approach is used by tools like Slab or ARMC. To model a system, transition constraint systems are used; they consist of a set of transitions which are first-order relations between a set of variables and the variables in the next state.
The intuitive user interface of Uppaal and the high number of models being available freely give rise to the idea to develop a method that automatically transforms Uppaal systems into transition constraint systems.
The straightforward approach to this would be to create the product automaton of the network of timed automata. This however leads to an explosion of the number of states.
So the goal of this thesis is to develop a method to translate the systems that avoids building the product automaton. The pivotal idea to this is to introduce a variable for each automaton that stores its location and to partition one step of the timed automata network into several steps of the transition system.

The thesis proceeds as following. In chapter 2 and 3 Uppaal and transition constraint systems are discussed. In chapter 4 the mathematical theory behind our conversion method is developed; for the most important results, formal proofs are given. Chapter 5 deals with several aspects of the implementation, while chapter 6 evaluates the implemented tool on a number of examples.

1.2 Related Work

Jochen Hoenicke showed in his PhD Thesis [Hoe06] how to convert phase event automata into transition constraint systems. In his approach, the product automaton of the phase event automata has to be constructed. Walid Haddad presented in his Master's Thesis [Had09] an alternative approach that avoided to construction of the product automaton by using substeps. This has some similarities to the approach used in this thesis to convert Uppaal systems.

2 Uppaal

Uppaal is a tool environment for modeling, simulating and verifying real-time systems developed by Uppsala University and Aalborg University. The systems are represented as networks of timed automata extended with several features like channel synchronization, integer variables and templates. To specify the properties to be checked, a subset of CTL ([HNSY92], [CE82]) is used.
The following sections first introduce the syntax and semantics of timed automata as presented in [BDL04] and [BY04]. After that, the most important additions are described.

2.1 Timed Automata

Timed automata ([AD90]) are finite-state machines extended with real-valued clock variables.
In the following definition, C denotes a set of clocks and $B(C)$ the set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C, c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.

Definition 1 (Timed Automaton (TA)). *A timed automaton is a tuple* $(L, l_0, \Sigma, C, E, I)$, *where*

- L *is a set of locations*
- $l_0 \in L$ *is the initial location*
- Σ *is a set of actions, co-actions and the internal* τ*-action*
- C *is a set of clocks*
- $E \subseteq L \times B(C) \times \Sigma \times 2^C \times L$ *the set of edges*
- $I : L \to B(C)$ *assigns invariants to locations*

For an edge $e = (l, g, a, r, l') \in E$, l is the start location, g is the guard (i.e. the condition that has to hold to take this edge), a is an action, r the set of clocks to be reset and l' the target location.

We will use $l \xrightarrow{g,a,r} l'$ to denote $(l, g, a, r, l') \in E$.

To define the semantics of a timed automaton, we use a clock valuation function $u : C \rightarrow \mathbb{R}_{\geq 0}$. Let $\mathbb{R}^C$ be the set of all clock valuations and let $u_0(x) = 0$ for all $x \in C$. For $d \in \mathbb{R}_+$, let $u + d$ be the clock valuation that maps all $x \in C$ to $u(x) + d$ and let $[r \mapsto 0]u$ denote the clock valuation that maps all clocks in r to 0 and agrees with u over $C \backslash r$. We write $u \in b$ with $b \in B(C)$ to mean that the clock values denoted by u satisfy b.

Definition 2 (Semantics of TA)**.** *Let $(L, l_0, \Sigma, C, E, I)$ be a timed automaton. The semantics is a labeled transition system $\langle S, s_0, \rightarrow \rangle$, where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state and $\rightarrow \subseteq S \times \{\mathbb{R}_{\geq 0} \cup \Sigma\} \times S$ is the transition relation such that:*

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

So there are two types of transitions. The first type is called a delay transition because the automaton can delay in a location for some time. The second type, in which the automaton follows an enabled edge, is called an action transition.

Definition 3 (Run of a TA)**.** *A run of a timed automaton $(L, l_0, \Sigma, C, E, I)$ with initial state (l_0, u_0) is a sequence of transitions $(l_0, u_0 + d_0) \xrightarrow{a_1}\xrightarrow{d_1} (l_1, u_1) \xrightarrow{a_2}\xrightarrow{d_2} ... \xrightarrow{a_n}\xrightarrow{d_n} (l_n, u_n)$ such that $u_0 + d_0 \in I(l_0)$, $d_i \in \mathbb{R}_{\geq 0}$ and $a_i \in \Sigma$*

Note that in the above definition, d_i is allowed to be 0, i.e. it is possible that between two successive action transition no time passes. Apart from that, it is also possible to contract two successive delay transitions with delays d_1 and d_2 to one delay transition with delay $d_1 + d_2$.

2.2 Networks of Timed Automata

Let a set of n timed automata $A_i = (L_i, l_i^0, \Sigma, C, E_i, I_i), 1 \leq i \leq n$ be given. These automata can be composed into a network of timed automata. We call $l = (l_1, ..., l_n)$ a location vector and combine the invariant functions into a common function over location vectors $I(l) = \bigwedge_i I_i(l_i)$. Let $l[l_i'/l_i]$ be the vector where the ith element of l is replaced by l_i'.

Definition 4 (Semantics of a network of Timed Automata)**.** *Let $A_i = (L_i, l_i^0, \Sigma, C, E_i, I_i)$ be a network of n timed automata and $l_0 = (l_1^0, ..., l_n^0)$ be the initial location vector. The semantics is a labeled transition system $\langle S, s_0, \rightarrow \rangle$, where $S = (L_1 \times ... \times L_n) \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state and $\rightarrow \subseteq S \times S$ is the transition relation such that:*

- $(l, u) \xrightarrow{d} (l, u + d)$ *if* $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(l)$
- $(l, u) \xrightarrow{\tau} (l[l_i'/l_i], u')$ *if there exists* $l_i \xrightarrow{g,\tau,r} l_i'$ *such that* $u \in g, u' = [r \mapsto 0]u$ *and* $u' \in I(l[l_i'/l_i])$
- $(l, u) \xrightarrow{c} (l[l_j'/l_j, l_i'/l_i], u')$ *if there exists* $i \neq j$ *such that*

    1. $l_i \xrightarrow{g_i,c?,r_i} l_i'$, $l_j \xrightarrow{g_j,c?,r_j} l_j'$, $u \in g_i \wedge g_j$, *and*
    2. $u' = [r_i \cup r_j \mapsto 0]u$ *and* $u' \in I(l[l_j'/l_j, l_i'/l_i])$

The rule for delay transitions is similar to the rule for single automata. For action transitions, there are two rules. The first allows an automaton to make a move on its own if the edge is not labeled with a channel. The second rule allows two automata to synchronize on a channel and move simultaneously.

2.3 Additional features in Uppaal

2.3.1 Broadcast Channels

Broadcast channels allow one-to-many synchronization. If an edge with a label $c!$ (with c being a broadcast channel) is taken, all other automata of the network that have an enabled edge labeled with $c?$ have to synchronize. Broadcast sending is not blocking, that means a sending edge with a broadcast channel can always fire, even if there are no receivers. On edges receiving on a broadcast channel, clock guards are not allowed.

2.3.2 Integer and boolean variables

In addition to real-valued clock variables, Uppaal also supports integer and boolean variables. The range of integer variables can be restricted by specifying upper and lower bounds; otherwise the default range of -32767 to 32767 is used.

2.3.3 Urgent and committed locations

As long as a process is in an urgent location, time is not allowed to pass. Committed locations further require that the next transition must involve an edge from one of the committed locations.

2.3.4 Other features

- Templates: Automata can be defined with a set of parameters and local variables. The parameters can be declared to have either call-by-value or call-by-reference semantics. They are substituted for a given argument in the process declaration.
- Urgent Channels: If a channel is declared as urgent, delays must not occur if a synchronization transition with an urgent channel is enabled.
- Arrays and structs

2.4 Expression in Uppaal

Classical timed automata only use simple expressions as guards and it is not possible to set a clock to a value different from 0. In contrast to that, Uppaal allows the use of expressions with the syntax given in Figure 2.1 .
To represent guards, only side-effect free expressions are allowed. Additionally, clocks and clock differences can only be compared to integer expressions and disjunctions are just allowed over integer conditions.
Synchronization labels are either of the form *Expression*! or *Expression*? or they are empty. The expressions must evaluate to a channel. Furthermore, they have to be side-effect free and may only use integers and channels (no clocks).
Instead of a set of clocks to be reset, edges in Uppaal use comma separated list of expressions. The expressions must have side-effects and to clocks only integer values can be assigned.
Invariants are expressions that are conjunctions of conditions of the form $x < e$ or $x \leq e$ where x is a clock and e a side-effect free expression evaluating to an integer.

2.5 Semantics of Uppaal systems

This section presents a pseudo-formal semantics for Uppaal systems. As in definition 4, it is again defined in terms of a labeled transition system

```
Expression ::= ID
            |  NAT
            |  Expression '[' Expression ']'
            |  '(' Expression ')'
            |  Expression '++' | '++' Expression
            |  Expression '--' | '--' Expression
            |  Expression Assign Expression
            |  Unary Expression
            |  Expression Binary Expression
            |  Expression '?' Expression ':' Expression
            |  Expression '.' ID
            |  Expression '(' Arguments ')'
            |  'forall' '(' ID ':' Type ')' Expression
            |  'exists' '(' ID ':' Type ')' Expression
            |  'deadlock' | 'true' | 'false'

Arguments  ::= [ Expression ( ',' Expression )* ]

Assign     ::= '=' | ':=' | '+=' | '-=' | '*=' | '/=' | '%='
            |  '|=' | '&=' | '^=' | '<<=' | '>>='
Unary      ::= '+' | '-' | '!' | 'not'
Binary     ::= '<' | '<=' | '==' | '!=' | '>=' | '>'
            |  '+' | '-' | '*' | '/' | '%' | '&'
            |  '|' | '^' | '<<' | '>>' | '&&' | '||'
            |  '<?' | '>?' | 'or' | 'and' | 'imply'
```

Figure 2.1: Syntax of expressions in Uppaal

$\langle S, s_0, \rightarrow \rangle$. We extend the clock valuation u to integer variables and define a function Inv that maps locations and location vectors to invariants.
There is a delay transition $(l, u) \rightarrow (l, u + d)$ if and only if

- $\forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in Inv(l)$
- l contains neither committed nor urgent locations
- for all locations $l_i \in l$ and for all locations l_j, if there is an edge from l_i to l_j then either
  - this edge does not synchronize over an urgent channel, or

  - this edge does synchronize over an urgent channel, but for all $0 \leq d' \leq d$ we have that $u + d'$ does not satisfy the guard of the edge.

As to action transitions, we have to distinguish three cases: internal transitions, binary synchronization and broadcast synchronization.
There is an internal transition $(l, u) \rightarrow (l', u')$ if there is an edge e from l_i to l'_i such that

- there is no synchronization label on e
- u satisfies the guard of e
- $l' = l[l'_i/l_i]$
- u' is obtained from u by executing the update label of e
- u' satisfies $Inv(l')$
- either l_i is committed or no other location in l is committed

As to binary synchronization, we have a transition $(l, u) \rightarrow (l', u')$ if there are two edges $e_1 = (l_1, g_1, a_1, r_1, l'_1)$ and $e_2 = (l_2, g_2, a_2, r_2, l'_2)$ in two different processes such that

- $a_1 = c!$ and $a_2 = c?$, where c is a binary channel
- u satisfies g_1 and g_2
- $l' = l[l'_1/l_1, l'_2/l_2]$
- u' is obtained from u by first executing the update label of e_1 and then the update label of e_2
- u' satisfies $Inv(l')$
- either
  - l_1 or l_2 or both locations are committed, or
  - no other location in l is committed

For broadcast synchronization we assume an order $p_1, p_2, ..., p_n$ of processes that is given by the order in which they are declared. There is a transition $(l, u) \rightarrow (l', u')$ if there is an edge $e = (l_0, g_0, a_0, r_0, l'_0)$ and m edges $e_i = (l_i, g_i, a_i, r_i, l'_i)$ for $1 \leq i \leq m$ such that

- $e, e_1, e_2, ..., e_m$ are in different processes
- $e_1, e_2, ..., e_m$ are ordered according to the ordering of the processes $p_1, p_2, ..., p_n$
- $a_0 = c!$ and $a_i = c?$ for all $1 \leq i \leq m$, where c is a broadcast channel
- u satisfies $g_0, g_1, g_2, ..., g_m$
- for all locations in l that are not a source of one of the edges $e, e_1, e_2, ..., e_m$, all edges from that location either do not have a synchronization label $c?$ or u does not satisfy the guard on the edge
- $l' = l[l'_0/l_0, l'_1/l_1, l'_2/l_2, ..., l'_m/l_m]$
- u' is obtained from u by first executing the update label of e and then the update labels of $e_1, ..., e_m$ (in this order)
- u' satisfies $Inv(l')$
- either
  - one or more of the locations $l, l_1, l_2, ..., l_m$ are committed, or
  - no other location in l is committed

2.6 Requirement Specification Language

To specify requirements that should be verified by the model checking engine of Uppaal, a simplified version of CTL is used. It has the following syntax:

```
Prop ::= 'E<>' Expression
       | 'A[]' Expression
       | 'E[]' Expression
       | 'A<>' Expression
       | Expression --> Expression
```

Note that in contrast to CTL, nested formulas are not allowed.

The property `E<> p` ("possibly p") evaluates to true if and only if there is a reachable state (in the transition system defined in the last section) in which p holds.
`A[] p` ("invariantly p") evaluates to true iff every reachable state satisfies p. It is equivalent to `not E<> not p`.

The property `E[] p` (”potentially always p”) evaluates to true iff there exists a path that is either infinite or the last state has no outgoing transitions in which p is always true. `A<> p` (”eventually p”) evaluates to true iff all possible transition sequences eventually reach a state satisfying p. It can be expressed as the property `not E[] not p`.
The ”leads to” property `p --> q` evaluates to true iff whenever p holds, q will eventually hold as well.
The expressions used in these formulas have to be side effect free. In addition to the expressions considered so far, it is also possible to test if a process is in a certain location by using an expression of the form *process.location*.

According to [BY04] the properties `E<> p` and `A[] p` are ”the two types of properties most commonly used in verification of timed systems” whereas ”the other three types [...] are not commonly used in Uppaal case-studies”.

2.7 Example

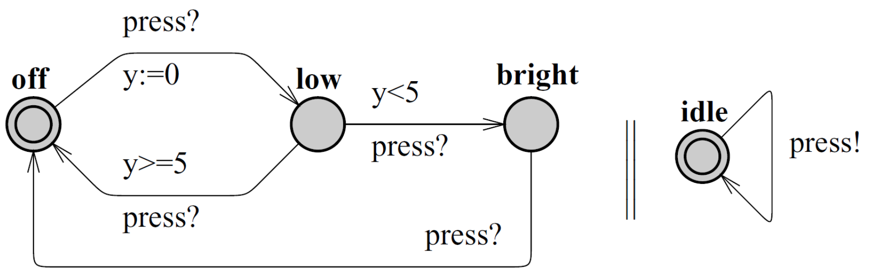
To illustrate the results of the previous sections, we will now take a look at the example in Figure 2.2 which is taken from [BDL04]. It consists of two processes: the first models a lamp, the second a user that controls the lamp. If the user presses a button, the two processes synchronize on the channel *press* which leads to the lamp being switched on. If the user presses the button again within the next 5 seconds, the lamp becomes brighter. If he presses the button again 5 or more seconds later, the lamp is turned off again. This is modeled by using a clock y that stores the time that has passed since the lamp was switched on.
We can now use Uppaal to verify that for example the location bright is reachable by checking the requirement `E<> Lamp.bright`.

2.8 Technical details

Uppaal uses XML-files to store the systems and files with the ending ”.q” for the requirements. Listing 2.1 shows the XML-file that corresponds to the example of the last section.

Listing 2.1: XML-file of the lamp example

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System 1.1//EN'
'http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd'>
<nta>
<declaration>
```



(a) Lamp. (b) User.

Figure 2.2: Simple lamp example

```
clock y;
chan press;
</declaration>
<template>
<name x="5" y="5">Lamp</name>
<declaration></declaration>
<location id="id0" x="-80" y="-176">
<name x="-90" y="-206">bright</name>
</location>
<location id="id1" x="-192" y="-176">
<name x="-202" y="-206">low</name>
</location>
<location id="id2" x="-312" y="-176">
<name x="-322" y="-206">off</name>
</location>
<init ref="id2"/>
<transition><source ref="id0"/><target ref="id2"/>
<label kind="synchronisation" x="-256" y="-191">press?</label>
</transition>
<transition><source ref="id1"/><target ref="id0"/>
<label kind="guard" x="-160" y="-200">y<5</label>
<label kind="synchronisation" x="-216" y="-80">press?</label>
</transition>
<transition><source ref="id1"/><target ref="id2"/>
<label kind="guard" x="-264" y="-136">y>=5</label>
</transition><transition><source ref="id2"/><target ref="id1"/>
<label kind="synchronisation" x="-272" y="-240">press?</label>
<label kind="assignment" x="-264" y="-216">y=0</label>
</transition>
</template>
<template><name>User</name>
<location id="id3" x="0" y="0">
<name x="-10" y="-30">idle</name>
```

```
</location>
<init ref="id3"/>
<transition><source ref="id3"/><target ref="id3"/>
<label kind="synchronisation" x="-60" y="-15">press!</label>
</transition>
</template>
<system>
system Lamp, User;
</system>
</nta>
```

3 Transition Constraint Systems

Tools like ARMC [RP07] and Slab [BDFW07] use transition constraint systems to represent the models. In this section, we will first introduce transition constraint systems. Then runs will be defined and we will consider an example.

3.1 Definition

Let $\mathcal{V}$ be a set of variables. A constraint over $\mathcal{V}$ is a formula of the form $v \bowtie e$ where $v \in \mathcal{V}$, $\bowtie \in \{<, \leq, =, \geq, >\}$ and e is an arithmetic expression containing the operators $+, -$ and $*$, variables from $\mathcal{V}$ and constant integers. We denote the set of conjunctions over constraints by Asrt($\mathcal{V}$).

Definition 5 (Transition Constraint Systems)**.** *A Transition Constraint System (TCS) $S = (V, init, error, T)$ consists of*

- *$V \subseteq \mathcal{V}$: a finite set of system variables; for each $v \in V$ a variable $v' \in V' \subseteq (\mathcal{V} \backslash V)$ is defined which indicates the value of v in the next state*
- *$init(V) \subseteq Asrt(V)$: the initial condition*
- *$error(V) \subseteq Asrt(V)$: the error condition*
- *$T \subseteq Asrt(V \cup V')$: the transition set; each transition is a conjunction $\bigwedge_i g_i(V) \wedge \bigwedge_i t_i(V, V')$ of guards g_i and transition relations t_i*

A state of a transition constraint system S is a valuation of the system variables V. A run of S is a sequence of states and transitions $s_0, \tau_0, s_1, \tau_1, ..., \tau_{n-1}, s_n$ such that s_0 satisfies *init* and for all $0 \leq i < n$ $\tau_i(s_i, s_{i+1})$ holds.

3.2 Example

We now consider an example that is somewhat similar to the lamp example in Section 2.7. This time, we model a lamp that might be used in a stairway. Apart from the possibility of turning the lamp brighter by pressing the button twice within 5 seconds, it should switch itself off after at most 60 seconds. The model uses discrete time steps with length 1 second. The transition constraint system S is given in Figure 3.2. Figure 3.1 models the same scenario as a Uppaal system (with *time* being an integer variable).

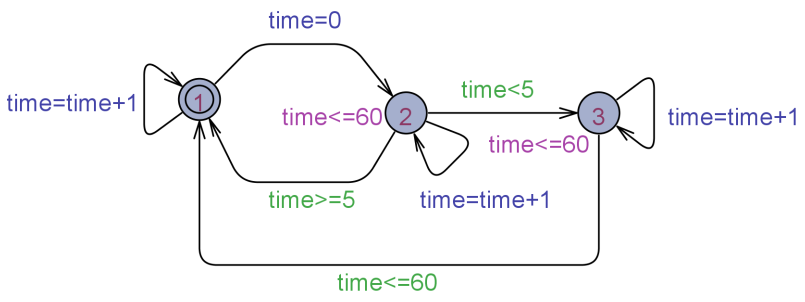


Figure 3.1: Uppaal model of the modified lamp example

Let $S = (V, init, error, T)$ be a transition constraint system such that

$$\begin{aligned}
V &= \{pc, time\}\\
init(V) &= (pc = 0 \land time = 0)\\
T &= \{pc = 0 \land pc' = pc \land time' = time + 1\\
&\quad pc = 0 \land pc' = 1 \land time' = 0,\\
&\quad pc = 1 \land time < 5 \land pc' = 2 \land time' = time,\\
&\quad pc = 1 \land time \geq 5 \land pc' = 0 \land time' = time,\\
&\quad pc = 1 \land time < 60 \land pc' = 1 \land time' = time + 1,\\
&\quad pc = 1 \land time \geq 60 \land pc' = 0 \land time' = time + 1,\\
&\quad pc = 2 \land time < 60 \land pc' = 2 \land time' = time + 1,\\
&\quad pc = 2 \land time \geq 60 \land pc' = 2 \land time' = time + 1\}\\
error(V) &= \{pc = 1 \land time > 60,\\
&\quad pc = 2 \land time > 60\}
\end{aligned}$$

Figure 3.2: TCS for the modified lamp example

4
Conversion

This chapter describes how Uppaal systems are translated into Transition Constraint Systems. First, only plain timed automata are considered and then the details like synchronization, urgent and committed location are added successively. The conversion of the error conditions is treated separately in section 4.6.
For reasons of readability, constraints of the form $a' = a$ are usually omitted.

4.1 Dealing with time

In this section, we will only consider single timed automata that don't use any synchronization. The translation process described here is similar to [Hoe06] who showed how to translate phase event automata into transition constraint systems.
To represent time, a variable *len* is introduced that denotes the time that has passed since the last transition. Apart from that, for each clock a corresponding variable is added and the variable *pc* determines the location the timed automaton currently is in.

In order to be able to prove that the converted transition constraint system is equivalent to the original automaton, we first have to define what we mean by equivalent.

Definition 6. *Let $A = (L, l_0, \Sigma, C, E, I)$ be a timed automaton and $S = (V, init, error, T)$ be a transition constraint system with $C \cup \{pc\} \subseteq V$. We define an equivalence $\approx \subseteq (L \times \mathbb{R}^C) \times \mathbb{R}^V$ between states of A and states of S, such that $(l_n, u_n) \approx s_m$ if*

- $s_m(pc) = l_n$
- $\forall c \in C : s_m(c) = u_n(c)$.

Definition 7. *We say that a timed automaton $A = (L, l_0, \Sigma, C, E, I)$ is equivalent to a TCS $S = (V, init, error, T)$ if and only if for every run of A there is a corresponding run of S and for every run of S there is a corresponding run of A. Two runs are corresponding if the last states of both are equivalent.*

Next, the rules how to translate a timed automaton into a TCS will be given.

Theorem 1. *Let $(L, l_0, \Sigma, C, E, I)$ be a timed automaton.*

An equivalent TCS $S = (V, init, error, T)$ is given by

$$\begin{aligned}
V &= C \cup \{pc, len\} \\
init(V) &= (pc = l_0 \land len \geq 0 \land (\bigwedge_{c \in C} c = len) \land I(l_0)) \\
T &= \{pc = l \land pc' = l' \land len' \geq 0 \land g \land \\
&\quad (\bigwedge_{c \in r} c' = len') \land (\bigwedge_{c \in C \setminus r} c' = c + len') \land I'(l') \\
&\quad |(l, g, a, r, l') \in E\}
\end{aligned}$$

The $init(V)$ formula demands that the automaton starts in its initial location. In addition to that, all clocks are set to the value of len which indicates the time spent in the initial location. This value has to be ≥ 0 and the invariant of the initial location must hold.
The formula T adds for each edge $e = (l, g, s, r, l') \in E$ a transition which requires that the pc variable is set to l and the guard is satisfied. The clocks that are reset on e are set to the time spent in l' (which corresponds to len') and the other clocks are increased by that amount. Again, len' can only take on values that satisfy the invariant of the new location.

Proof. Let A=$(L, l_0, \Sigma, C, E, I)$ be a timed automaton and S=$(V, init, error, T)$ the TCS constructed as stated in the theorem.

"$\Rightarrow$"
Let $(l_0, u_0 + d_0) \xrightarrow{a_1}\xrightarrow{d_1} (l_2, u_2) \xrightarrow{a_2}\xrightarrow{d_2} ... \xrightarrow{a_n}\xrightarrow{d_n} (l_n, u_n)$ be a run of A. We have to show that there is a corresponding run of T. The proof is by induction on the length of the run of A.

Induction base: n=0
For $(l_0, u_0 + d_0)$ the corresponding run is s_0 such that

- $s_0(pc) = l_0$

- $s_0(len) = d_0$
- $\forall c \in C : s_0(c) = (u_0 + d_0)(c) = s_0(len)$

Since u_0+d_0 satisfies $I(l_0)$ it follows that s_0 satisfies $I(l_0)$ and thus s_0 satisfies $init$.

Induction step: $n - 1 \rightarrow n$
By the induction hypothesis, there is a run of T that is corresponding to $(l_0, u_0+d_0) \xrightarrow{a_1}\xrightarrow{d_1} (l_1, u_1) \xrightarrow{a_2}\xrightarrow{d_2} ... \xrightarrow{a_{n-1}}\xrightarrow{d_{n-1}} (l_{n-1}, u_{n-1})$. Let $s_0, \tau_0, s_1, \tau_1, ..., \tau_{m-1}, s_m$ be this run.

Since $(l_{n-1}, u_{n-1}) \xrightarrow{a_n}\xrightarrow{d_n} (l_n, u_n)$, there exists an edge $e = (l_{n-1}, g, a_n, r, l_n) \in E$ such that $u_{n-1} \in g, u_n = ([r \mapsto 0]u_{n-1}) + d_n$ and $u_n \in I(l_n)$. Let s_{m+1} be a valuation such that

- $s_{m+1}(pc) = l_n$
- $s_{m+1}(len) = d_n$
- $\forall c \in r : s_{m+1}(c) = s_{m+1}(len) = d_n = u_n(c)$
- $\forall c \in C \backslash r : s_{m+1}(c) = s_m(c) + s_{m+1}(len) = u_{n-1}(c) + d_n = u_n(c)$
- $s_{m+1}(y) = s_m(y)$ for all other variables y

So $(l_n, u_n) \approx s_{m+1}$. Since $(l_{n-1}, u_{n-1}) \xrightarrow{a_n}\xrightarrow{d_n} (l_n, u_n)$ implies that g is satisfied and also $I(l_n)$ is satisfied, $\tau_m(s_m, s_{m+1})$ holds and thus $s_0, \tau_0, s_1, \tau_1, ..., s_m, \tau_m, s_{m+1}$ is a run of S that is equivalent to the run of A.

"$\Leftarrow$"
Let $s_0, \tau_0, s_1, \tau_1, ..., \tau_{n-1}, s_n$ be a run of a S. We have to show that there is a corresponding run of A. The proof is again by induction.

Induction base: n=0
For s_0 the corresponding run is $(l_0, u_0 + d_0)$ such that

- $l_0 = s_0(pc)$
- $d_0 = s_0(len)$
- $\forall c \in C : (u_0 + d_0)(c) = d_0 = s_0(len) = s_0(c)$

Since s_0 satisfies $I(l_0)$ the two states are equivalent.

Induction step: $n-1 \rightarrow n$
By the induction hypothesis, there is a run of A that is corresponding to $s_0, \tau_0, s_1, \tau_1, ..., \tau_{n-2}, s_{n-1}$. Let $(l_0, u_0 + d_0) \xrightarrow{a_1}\xrightarrow{d_1} (l_1, u_1) \xrightarrow{a_2}\xrightarrow{d_2} ... \xrightarrow{a_m}\xrightarrow{d_m} (l_m, u_m)$ be this run.

Since $\tau_{n-1}(s_{n-1}, s_n)$ holds, there must be an edge $e = (l, g, a, r, l') \in E$ such that

- $l = l_m = s_{n-1}(pc)$
- $l' = s_n(pc)$
- s_{n-1} satisfies g
- $a \in \Sigma$
- $\forall c \in r : s_n(c) = s_n(len)$
- $\forall c \in C \backslash r : s_n(c) = s_{n-1}(c) + s_n(len)$
- s_n satisfies $I(l')$

But then $(l_m, u_m) \xrightarrow{a}\xrightarrow{s_n(len)} (l', ([r \mapsto 0]u_m) + s_n(len))$. Since $(l', ([r \mapsto 0]u_m) + s_n(len)) \approx s_n$ there is a corresponding run and thus A and S are equivalent.

$\square$

4.2 Networks of timed automata

Next, we consider a system of n parallel timed automata that are not allowed to use synchronization. For each automaton, we add a variable (pc_i) that denotes the current location. In addition to that, a variable *substate* is introduced. It allows for partitioning one step of the timed automata system into several steps of the corresponding transition constraint system.

We first extend the definition of equivalence to networks of timed automata.

Definition 8. *Let $A_i = (L_i, l_i^0, \Sigma, C, E_i, I_i), 1 \leq i \leq n$ be a a network of n timed automata and $S = (V, init, error, T)$ be a transition constraint system with $C \cup \{pc_i | 1 \leq i \leq n\} \cup \{substate\} \subseteq V$. We define an equivalence $\approx \subseteq (L^n \times \mathbb{R}^C) \times \mathbb{R}^V$ between states of A and states of S, such that $((l_1, l_2, ..., l_n), u_x) \approx s_y$ if*

- $s_y(pc_1) = l_1 \wedge s_y(pc_2) = l_2 \wedge ... \wedge s_y(pc_n) = l_n$
- $\forall c \in C : s_y(c) = u_x(c)$.
- $s_y(substate) = 0$.

To establish an equivalence between runs of a network of timed automata and runs of transitions constraint systems, we introduce the notion of a *complete run*, which is how we will call a run that ends in a state were the substate variable is 0.

Definition 9. *A complete run of a TCS $S = (V, init, error, T)$ with $substate \in V$ is a sequence of states and transitions $s_0, \tau_0, s_1, \tau_1, ..., \tau_{n-1}, s_n$ such that s_0 satisfies init, for all $0 \leq i < n$ $\tau_i(s_i, s_{i+1})$ holds and $s_n(substate) = 0$.*

We are now able to define an equivalence between runs of networks of timed automata and complete runs of transition constraint systems.

Definition 10. *We say that a network of timed automaton $A_i = (L_i, l_i^0, \Sigma, C, E_i, I_i)$ is equivalent to a TCS $S = (V, init, error, T)$ if and only if for every run of A_i there is a corresponding complete run of S and for every complete run of S there is a corresponding run of A_i. Two runs are corresponding if the last states of both are equivalent.*

The following theorem describes how the conversion is carried out. In order to facilitate reading, terms like $pc_1 = l_1^0 \wedge ... \wedge pc_n = l_n^0$ are replaced by $pc_a = l_a^0$ with a being a free variable.

Theorem 2. *Let $A_i = (L_i, l_i^0, \Sigma, C, E_i, I_i)$ be a network of n timed automata.*

An equivalent TCS $S = (V, init, error, T)$ is given by

$$\begin{aligned}
V &= C \cup \{pc_i | 1 \leq i \leq n\} \cup \{len, substate\} \\
init(V) &= (pc_a = l_a^0 \wedge substate = 0 \wedge len \geq 0 \wedge (\bigwedge_{c \in C} c = len) \wedge I_a(l_a^0)) \\
T &= \{pc_i = l \wedge substate = 0 \wedge pc_i' = l' \wedge len' \geq 0 \wedge g \wedge \\
&\quad (\bigwedge_{c \in r} c' = len') \wedge (\bigwedge_{c \in C \setminus r} c' = c + len') \\
&\quad \wedge substate' = chInv_1 | (l, g, \tau, r, l') \in E_i, 1 \leq i \leq n\} \\
&\cup \{substate = chInv_i \wedge pc_i = l \wedge I_i(l) \wedge substate' = chInv_{i+1} \\
&\quad | 1 \leq i \leq n-1, l \in L_i\} \\
&\cup \{substate = chInv_n \wedge pc_n = l \wedge I_n(l) \wedge substate' = 0 | l \in L_n\}
\end{aligned}$$

The $init(V)$ formula demands that all automata start in their initial locations and that the invariants of the initial locations hold with the clocks being set to len. Furthermore, the substate variable is set to 0.
For each edge $e = (l, g, \tau, r, l') \in E_i$ a transition is added that updates the pc variable of automaton i from l to l' if the guard is satisfied and the substate variable is 0. The clocks are increased by a value ≥ 0 and the substate variable is set to $chInv_1$.
For each location l in automaton i, a transition is added that requires the pc of i to be in l, the invariant of l to be satisfied and the substate variable to be set to $chInv_i$. The substate variable is updated to $chInv_{i+1}$ for $1 \leq i < n$ and to 0 for $i = n$.
So for each transition of an automaton, all automata have to check if their invariant is still satisfied. If this is the case, the substate variable is set to 0 again and the next edge can be taken. If the invariant is not satisfied for an automaton, the TCS will deadlock. Regarding equivalence, this is however not a problem since a state of network of automata can only be equivalent to a state of the TCS if the substate variable is set to 0 (see Definition 8).

We will now prove the correctness of Theorem 2.

Proof. Let $A_i = (L_i, l_i^0, \Sigma, C, E_i, I_i)$ be a network of n timed automata and S=$(V, init, error, T)$ the TCS constructed as stated in the theorem.

"$\Rightarrow$"
Let $(l_0, u_0 + d_0) \xrightarrow{a_1} \xrightarrow{d_1} (l_2, u_2) \xrightarrow{a_2} \xrightarrow{d_2} \ldots \xrightarrow{a_n} \xrightarrow{d_n} (l_n, u_n)$ be a run of A. We have to show that there is a corresponding complete run of T. The proof is by induction on the length of the run of A.

Induction base: n=0
For $(l_0, u_0 + d_0)$ with $l_0 = (l_1^0, l_2^0, ..., l_n^0)$ the corresponding run is s_0 such that

- $s_0(pc_1) = l_1^0$, $s_0(pc_2) = l_2^0$, ..., $s_0(pc_n) = l_n^0$
- $s_0(len) = d_0$
- $\forall c \in C : s_0(c) = (u_0 + d_0)(c) = s_0(len)$

Since $u_0 + d_0$ satisfies $I(l_0)$ it follows that $u_0 + d_0$ satisfies $I(l_1^0) \wedge ... \wedge I(l_n^0)$. Thus s_0 satisfies satisfies $init$.

Induction step: $n - 1 \rightarrow n$
By the induction hypothesis, there is a run of T that is corresponding to

$(l_0, u_0{+}d_0) \xrightarrow{a_1}\xrightarrow{d_1} (l_1, u_1) \xrightarrow{a_2}\xrightarrow{d_2} ... \xrightarrow{a_{n-1}}\xrightarrow{d_{n-1}} (l_{n-1}, u_{n-1})$. Let $s_0, \tau_0, s_1, \tau_1, ..., \tau_{m-1}, s_m$ be this run.

Since $(l_{n-1}, u_{n-1}) \xrightarrow{a_n}\xrightarrow{d_n} (l_n, u_n)$ there exists an edge $e = (l_i^{n-1}, g, \tau, r, l_i^n) \in E_i$ such that $l_n = l_{n-1}[l_i^n/l_i^{n-1}], u_{n-1} \in g, u_n = ([r \mapsto 0]u_{n-1}){+}d_n$ and $u_n \in I(l_n)$. Let s_{m+1} be a valuation such that

- $s_{m+1}(pc_i) = l_i^n$
- $s_{m+1}(len) = d_n$
- $\forall c \in r : s_{m+1}(c) = s_{m+1}(len) = d_n = u_n(c)$
- $\forall c \in C\backslash r : s_{m+1}(c) = s_m(c) + s_{m+1}(len) = u_{n-1}(c) + d_n = u_n(c)$
- $s_{m+1}(substate) = chInv_1$
- $s_{m+1}(y) = s_m(y)$ for all other variables y

Since $(l_{n-1}, u_{n-1}) \xrightarrow{a_n}\xrightarrow{d_n} (l_n, u_n)$ implies that g is satisfied, $\tau_m(s_m, s_{m+1})$ holds. Let $s_{m+1+k}, 1 \leq k \leq n$ be a valuation such that

- $s_{m+1+k}(pc_i) = l_k^n$
- $s_{m+1}(substate) = chInv_{k+1}$ if $k < n$
- $s_{m+1}(substate) = 0$ if $k = n$
- $s_{m+1}(y) = s_m(y)$ for all other variables y

Since $u_n \in I(l_n)$ it follows that $\forall 1 \leq k \leq n : s_{m+1+k} \in I_k(l_k^n)$ and thus $\tau_{m+1+k}(s_{m+1+k}, s_{m+1+k+1})$ holds for $1 \leq k < n$.
$(l_n, u_n) \approx s_{m+1+n}$ and thus $s_0, \tau_0, s_1, \tau_1, ..., s_{m+n}, \tau_{m+n}, s_{m+1+n}$ is a complete run of S that is equivalent to the run of A.

"$\Leftarrow$"
Let $s_0, \tau_0, s_1, \tau_1, ..., \tau_{n-1}, s_n$ be a complete run of a S. We have to show that there is a corresponding run of A_i. The proof is again by induction.

Induction base: n=0
For s_0 the corresponding run is $((l_1^0, ..., l_n^0), u_0 + d_0)$ such that

- $l_1^0 = s_0(pc_1) \wedge l_2^0 = s_0(pc_2) \wedge ... \wedge l_n^0 = s_0(pc_n)$
- $d_0 = s_0(len)$

- $\forall c \in C : (u_0 + d_0)(c) = d_0 = s_0(len) = s_0(c)$

Since s_0 satisfies $I_1(l_1^0) \wedge ... \wedge I_n(l_n^0)$ and $s_0(substate) = 0$ the two states are equivalent.

Induction step: $n - 1 \rightarrow n$
Since $s_0, \tau_0, s_1, \tau_1, ..., \tau_{p-1}, s_p$ is a complete run of S, $s_p(substate) = 0$. Let $s_q, 0 \leq q < p$ be the state such that $s_q(substate) = 0$ and for all $q < i < p$ $s_i(substate) \neq 0$. By the induction hypothesis, there is a run of A_i that is corresponding to $s_0, \tau_0, s_1, \tau_1, ..., \tau_{q-1}, s_q$. Let $(l_0, u_0+d_0) \xrightarrow{a_1}\xrightarrow{d_1} (l_1, u_1) \xrightarrow{a_2}\xrightarrow{d_2} ... \xrightarrow{a_m}\xrightarrow{d_m} ((l_1^m, ..., l_n^m), u_m)$ be this run.

Since $\tau_q(s_q, s_{q+1})$ (with $s_q(substate) = 0$) holds, there must be an edge $e = (l, g, \tau, r, l') \in E_i$ for some $1 \leq i \leq n$ such that

- $s_{q+1}(substate) = chInv_1$
- $l = l_i^m = s_q(pc_i)$
- $l' = s_{q+1}(pc)$
- s_q satisfies g
- $\forall c \in r : s_{q+1}(c) = s_{q+1}(len)$
- $\forall c \in C \backslash r : s_{q+1}(c) = s_q(c) + s_{q+1}(len)$

Since $\tau_{q+j}(s_{q+j}, s_{q+1+j}), 1 \leq j < p - q$ it follows from the construction that $p - q = n$ and

- $s_{q+j}(substate) = chInv_j$ if $j < n$
- $s_p(substate) = 0$
- $s_{q+j}(pc) = l_j^m$ if $j \neq i$
- $s_{q+j}(pc) = l'$ if $j \neq i$
- s_{q+j} satisfies $I_j(l_j^m)$

Since by construction $\forall c \in C : s_{q+1}(c) = s_{q+2}(c) = ... = s_p(c)$ it follows that $((l_1^m, ..., l_n^m), u_m) \xrightarrow{\tau}\xrightarrow{s_{q+1}(len)} ((l_1^m, ..., l', ..., l_n^m), ([r \mapsto 0]u_m) + s_{q+1}(len))$. Since $((l_1^m, ..., l', ..., l_n^m), ([r \mapsto 0]u_m) + s_{q+1}(len)) \approx s_p$ there is a corresponding run and thus A and S are equivalent.

□

4.3 Binary Synchronization

Now we consider networks that are allowed to use binary synchronization. We will use a function $num : (\bigcup_i E_i) \to \mathbb{N}$ that assigns unique natural numbers to edges. The new variables e_s and e_r are used to store the send and receive edge during a synchronization process.

Theorem 3. *Let $A_i = (L_i, l_i^0, \Sigma, C, E_i, I_i)$ be a network of n timed automata.*

An equivalent TCS $S = (V, init, error, T)$ is given by

$$
\begin{aligned}
V &= C \cup \{pc_i | 1 \le i \le n\} \cup \{len, substate, e_s, e_r\}\\
init(V) &= (pc_a = l_a^0 \wedge I_a(l_a^0) \wedge len \ge 0 \wedge (\bigwedge_{c \in C} c = len) \wedge substate = 0 \wedge\\
&\quad e_s = 0 \wedge e_r = 0)\\
T &= \{pc_i = l \wedge substate = 0 \wedge pc_i' = l' \wedge len' \ge 0 \wedge g \wedge\\
&\quad (\bigwedge_{c \in r} c' = len') \wedge (\bigwedge_{c \in C \setminus r} c' = c + len') \wedge substate' = chInv_1\\
&\quad |(l, g, s, r, l') \in E_i, 1 \le i \le n, s = \tau\}\\
&\quad \cup \{substate = chInv_i \wedge pc_i = l \wedge I_i(l) \wedge substate = chInv_{i+1}\\
&\quad |1 \le i \le n-1, l \in L_i\}\\
&\quad \cup \{substate = chInv_n \wedge pc_n = l \wedge I_n(l) \wedge substate = 0 | l \in L_n\}\\
&\quad \cup \{pc_i = l \wedge substate = 0 \wedge g \wedge substate' = rec_c \wedge e_s' = num(e)\\
&\quad |e \in E_i, e = (l, g, s, r, l'), s = c!, c \in \Sigma\}\\
&\quad \cup \{pc_i = l \wedge substate = rec_c \wedge g \wedge substate' = update_{send} \wedge e_r' = num(e)\\
&\quad |e \in E_i, e = (l, g, s, r, l'), s = c?, c \in \Sigma, num^{-1}(e_s) \notin E_i\}\\
&\quad \cup \{pc_i = l \wedge substate = update_{send} \wedge pc_i' = l' \wedge e_s = num(e)\\
&\quad \wedge substate' = update_{rec} \wedge \bigwedge_{c \in r} c' = 0\\
&\quad |e \in E_i, e = (l, g, s, r, l'), s = c!, c \in \Sigma\}\\
&\quad \cup \{pc_i = l \wedge substate = update_{rec} \wedge pc_i' = l' \wedge e_r = num(e)\\
&\quad \wedge len' \ge 0 \wedge substate' = chInv_1 \wedge (\bigwedge_{c \in r} c' = len') \wedge\\
&\quad (\bigwedge_{c \in C \setminus r} c' = c + len') | e \in E_i, e = (l, g, s, r, l'), s = c?, c \in \Sigma\}
\end{aligned}
$$

The $init(V)$ formula is similar to the last section. The set of transition formulas has been extended with formulas for binary synchronization.
For each sending edge $e = (l, g, c!, r, l') \in E_i$, a transition that updates the substate from 0 to rec_c and stores the number of e in e_s if the guard is satisfied and pc_i is set to l, is added. Similarly, for each receiving edge $e = (l, g, c?, r, l') \in E_j$ a transition that update the substate from rec_c to

$update_{send}$ and stores the number of e in e_r is added. The transition requires that the receiving edge is in a different process than the sending edge. Futhermore, the guard of the edge has to be satisfied and pc_j needs to be set to l.
Additionally, for each sending and receiving edge, transitions that handle the update expressions are introduced. According to the semantics, the updates on the sending edges are executed before the updates on the receiving edges. Afterwards, the invariants are checked in the same fashion as dicussed in the previous section.

4.4 Urgent locations

Now we will consider network that are allowed to have urgent locations. We therefore introduce a variable u that denotes the number of processes that are currently in an urgent location. Apart from that, we define a function $ur : L \to 0, 1$ such that $ur(l) = 1$ if and only if l is an urgent location.

Theorem 4. *Let $T_i = (L_i, l_i^0, \Sigma, C, E_i, I_i)$ be a network of n timed automata.*

A corresponding TCS $S = (V, init, error, T)$ is given by

$$\begin{aligned}
V &= C \cup \{pc_i | 1 \leq i \leq n\} \cup \{len, substate, e_s, e_r, u\} \\
init(V) &= (pc_a = l_a^0 \wedge I_a(l_a^0) \wedge len \geq 0 \wedge (\bigwedge_{c \in C} c = len) \wedge substate = 0 \wedge \\
&\quad e_s = 0 \wedge e_r = 0 \wedge u = \sum_{j=1}^{n} ur(l_j^0)) \\
T &= \{pc_i = l \wedge substate = 0 \wedge pc_i' = l' \wedge g \wedge substate' = chInv_1 \wedge \\
&\quad (\bigwedge_{c \in r} c' = 0) \wedge (\bigwedge_{c \in C \setminus r} c' = c) \wedge ur(l') - ur(l) + u > 0 \\
&\quad \wedge u' = ur(l') - ur(l) + u | (l, g, s, r, l') \in E_i, 1 \leq i \leq n, s = \tau\} \\
&\quad \cup \{pc_i = l \wedge substate = 0 \wedge pc_i' = l' \wedge g \wedge substate' = chInv_1 \wedge \\
&\quad (\bigwedge_{c \in r} c' = len') \wedge (\bigwedge_{c \in C \setminus r} c' = c + len') \wedge ur(l') - ur(l) + u = 0 \wedge \\
&\quad len' \geq 0 \wedge u' = ur(l') - ur(l) + u \\
&\quad | (l, g, s, r, l') \in E_i, 1 \leq i \leq n, s = \tau\} \\
&\quad \cup \{substate = chInv_i \wedge pc_i = l \wedge I_i(l) \wedge substate = chInv_{i+1} \\
&\quad | 1 \leq i \leq n - 1, l \in L_i\} \\
&\quad \cup \{substate = chInv_n \wedge pc_n = l \wedge I_n(l) \wedge substate = 0 | l \in L_n\} \\
&\quad \cup \{pc_i = l \wedge substate = 0 \wedge g \wedge substate' = rec_c \wedge e_s' = num(e) \\
&\quad | e \in E_i, e = (l, g, s, r, l'), s = c!, c \in \Sigma\} \\
&\quad \cup \{pc_i = l \wedge substate = rec_c \wedge g \wedge substate' = update_{send} \wedge e_r' = num(e) \\
&\quad | e \in E_i, e = (l, g, s, r, l'), s = c?, c \in \Sigma, num^{-1}(e_s) \notin E_i\} \\
&\quad \cup \{pc_i = l \wedge substate = update_{send} \wedge pc_i' = l' \wedge e_s = num(e) \\
&\quad \wedge substate' = update_{rec} \wedge \bigwedge_{c \in r} c' = 0 \wedge u' = ur(l') - ur(l) + u \\
&\quad | e \in E_i, e = (l, g, s, r, l'), s = c!, c \in \Sigma\} \\
&\quad \cup \{pc_i = l \wedge substate = update_{rec} \wedge pc_i' = l' \wedge e_r = num(e) \\
&\quad \wedge substate' = chInv_1 \wedge (\bigwedge_{c \in r} c' = 0) \wedge (\bigwedge_{c \in C \setminus r} c' = c) \\
&\quad \wedge ur(l') - ur(l) + u > 0 \wedge u' = ur(l') - ur(l) + u
\end{aligned}$$

$$
\begin{aligned}
&|e \in E_i, e = (l, g, s, r, l'), s = c?, c \in \Sigma\} \\
\cup\, &\{pc_i = l \wedge substate = update_{rec} \wedge pc'_i = l' \wedge e_r = num(e) \\
&\wedge substate' = chInv_1 \wedge (\bigwedge_{c \in r} c' = len') \wedge (\bigwedge_{c \in C \setminus r} c' = c + len') \\
&\wedge len' \geq 0 \wedge ur(l') - ur(l) + u = 0 \wedge u' = ur(l') - ur(l) + u \\
&|e \in E_i, e = (l, g, s, r, l'), s = c?, c \in \Sigma\}
\end{aligned}
$$

4.5 Commited locations

To deal with committed locations we introduce a variable co that stores the number of processes currently being in a committed location. Additionally we define a function $co : L \to 0, 1$ such that $co(l) = 1$ if and only if l is a committed location
Note that $\forall l \in L : co(l) \Rightarrow ur(l)$

Theorem 5. *Let $_i = (L_i, l_i^0, \Sigma, C, E_i, I_i)$ be a network of n timed automata.*

A corresponding TCS $S = (V, init, error, T)$ is given by

$$\begin{aligned}
V &= C \cup \{pc_i | 1 \le i \le n\} \cup \{len, substate, e_s, e_r, u, co\} \\
init(V) &= (pc_a = l_a^0 \wedge I_a(l_a^0) \wedge len \ge 0 \wedge (\bigwedge_{c \in C} c = len) \wedge substate = 0 \wedge \\
&\quad e_s = 0 \wedge e_r = 0 \wedge u = 0 \wedge co = 0) \\
T &= \{pc_i = l \wedge substate = 0 \wedge pc_i' = l' \wedge g \wedge substate' = chInv_1 \wedge \\
&\quad (\bigwedge_{c \in r} c' = 0) \wedge (\bigwedge_{c \in C \setminus r} c' = c) \wedge ur(l') - ur(l) + u > 0 \wedge co(l) * co = co \\
&\quad \wedge u' = ur(l') - ur(l) + u \wedge co' = co(l') - co(l) + co | (l, g, s, r, l') \in E_i, 1 \le i \le n, s = \tau\} \\
&\quad \cup \{pc_i = l \wedge substate = 0 \wedge pc_i' = l' \wedge g \wedge substate' = chInv_1 \wedge \\
&\quad (\bigwedge_{c \in r} c' = len') \wedge (\bigwedge_{c \in C \setminus r} c' = c + len') \wedge ur(l') - ur(l) + u = 0 \wedge \\
&\quad len' \ge 0 \wedge u' = 0 \wedge co' = 0 \\
&\quad | (l, g, s, r, l') \in E_i, 1 \le i \le n, s = \tau\} \\
&\quad \cup \{substate = chInv_i \wedge pc_i = l \wedge I_i(l) \wedge substate = chInv_{i+1} \\
&\quad | 1 \le i \le n - 1, l \in L_i\} \\
&\quad \cup \{substate = chInv_n \wedge pc_n = l \wedge I_n(l) \wedge substate = 0 | l \in L_n\} \\
&\quad \cup \{pc_i = l \wedge substate = 0 \wedge g \wedge substate' = rec_c \wedge e_s' = num(e) \\
&\quad \wedge co(l) * co = co | e \in E_i, e = (l, g, s, r, l'), s = c!, c \in \Sigma\} \\
&\quad \cup \{pc_i = l \wedge substate = rec_c \wedge g \wedge substate' = update_{send} \wedge e_r' = num(e) \\
&\quad \wedge co(l) * co = co | e \in E_i, e = (l, g, s, r, l'), s = c?, c \in \Sigma, num^{-1}(e_s) \notin E_i\} \\
&\quad \cup \{pc_i = l \wedge substate = update_{send} \wedge pc_i' = l' \wedge e_s = num(e) \\
&\quad \wedge substate' = update_{rec} \wedge \bigwedge_{c \in r} c' = 0 \wedge u' = ur(l') - ur(l) + u \wedge co' = co(l') - co(l) + co \\
&\quad | e \in E_i, e = (l, g, s, r, l'), s = c!, c \in \Sigma\} \\
&\quad \cup \{pc_i = l \wedge substate = update_{rec} \wedge pc_i' = l' \wedge e_r = num(e) \\
&\quad \wedge substate' = chInv_1 \wedge (\bigwedge_{c \in r} c' = 0) \wedge (\bigwedge_{c \in C \setminus r} c' = c) \\
&\quad \wedge ur(l') - ur(l) + u > 0 \wedge u' = ur(l') - ur(l) + u \wedge co' = co(l') - co(l) + co \\
&\quad | e \in E_i, e = (l, g, s, r, l'), s = c?, c \in \Sigma\}
\end{aligned}$$

$$\begin{aligned}
&\cup \{pc_i = l \wedge substate = update_{rec} \wedge pc_i' = l' \wedge e_r = num(e)\\
&\quad \wedge substate' = chInv_1 \wedge (\bigwedge_{c \in r} c' = len') \wedge (\bigwedge_{c \in C \setminus r} c' = c + len')\\
&\quad \wedge len' \geq 0 \wedge ur(l') - ur(l) + u = 0 \wedge u' = 0 \wedge co' = 0\\
&\quad | e \in E_i, e = (l, g, s, r, l'), s = c?, c \in \Sigma\}
\end{aligned}$$

Note that $co(l) * co = co$ evaluates to 0 iff. $co(l) = 0$ and $co \neq 0$

4.6 Converting Requirement Specifications

In all conversions in the previous sections, one part of the transition constraint systems was left out: the error conditions. They are obtained from converting a requirement specification. We will show how to do this for "possibly" and "invariantly" properties.
For a "possibly" property `E<> p` we get the following error condition:

$$error(V) = \{trans(p)\}$$

. $trans(p)$ is obtained from p by replacing any expression of the form $A_i.l$ by a constraint of the form $pc_i = l$.
Note that the property `E<> p` is satisfied if and only if the error condition of the transition constraint system is satisfiable.

For an "invariantly" property `A[] p` we get the following error condition:

$$error(V) = \{\neg trans(p)\}$$

In this case, the property `A[] p` is satisfied if and only if the error condition is not satisfiable.

4.7 Example

We will now use the results of the previous sections to convert the Uppaal system introduced in Section 2.7 to a transition constraint system $S = (V, init, error, T)$.
Since the system consists of two processes and a clock y we get

$$V = \{pc_1, pc_2, y, len, substate, e_s, e_r\}$$

(we use pc_1 for the pc of process Lamp and pc_2 for the pc of process User). We leave out the variables for urgent and committed locations because the

$substate = 0 \land pc_2 = idle \land substate' = rec_{press} \land e'_s = 5$
$substate = rec_{press} \land pc_1 = high \land e_s > 4 \land substate' = update_{send} \land e'_r = 1$
$substate = rec_{press} \land pc_1 = low \land e_s > 4 \land y < 5 \land substate' = update_{send} \land e'_r = 2$
$substate = rec_{press} \land pc_1 = low \land e_s > 4 \land y \geq 5 \land substate' = update_{send} \land e'_r = 3$
$substate = rec_{press} \land pc_1 = off \land e_s > 4 \land substate' = update_{send} \land e'_r = 4$
$substate = update_{send} \land pc_2 = idle \land e_s = 5 \land substate' = update_{rec}$
$substate = update_{rec} \land pc_1 = high \land e_r = 1 \land substate' = chInv_1 \land pc'_1 = off \land len' \geq 0 \land y' = y + len'$
$substate = update_{rec} \land pc_1 = low \land e_r = 2 \land substate' = chInv_1 \land pc'_1 = high \land len' \geq 0 \land y' = y + len'$
$substate = update_{rec} \land pc_1 = low \land e_r = 3 \land substate' = chInv_1 \land pc'_1 = off \land len' \geq 0 \land y' = y + len'$
$substate = update_{rec} \land pc_1 = off \land e_r = 4 \land substate' = chInv_1 \land pc'_1 = low \land len' \geq 0 \land y' = len'$
$substate = update_{chInv_1} \land pc_1 = off \land substate' = chInv_2$
$substate = update_{chInv_1} \land pc_1 = low \land substate' = chInv_2$
$substate = update_{chInv_1} \land pc_1 = high \land substate' = chInv_2$
$substate = update_{chInv_2} \land pc_2 = idle \land substate' = 0$

Figure 4.1: Transition set for the lamp example

example does not contain urgent or committed locations.

The initial condition is

$$init(V) = (pc_1 = off \land pc_2 = idle \land len \geq 0 \land y = len \land substate = 0 \land e_r = 0 \land e_s = 0)$$

.

The error condition for the property `E<> Lamp.bright` is

$$error(V) = \{pc_1 = bright\}$$

The transition set is given in table 4.1. It uses the assumption that $num(e) < 5$ for the edges of process $Lamp$ and $num(e) = 5$ for the edge of process $User$.

5 Implementation

Using the results from chapter 4 we have implemented a tool that allows for converting Uppaal models automatically into corresponding models for ARMC/Slab. The tool has been written in Java, employing JJTree/JavaCC to generate the parser for expressions and declarations, JDOM for XML-parsing and JUnit for testing the individual parts. To achieve modularity and reusability, the system has been separated into three main parts, each of which has been implemented as a package in Java. The first package deals with everything that is related to Uppaal while the second deals with transition constraint systems. The third package constitutes the link between the first and the second, transforming a Uppaal system into a TCS. The following sections discuss the details of these packages; the corresponding class diagrams are given in Appendix A.

5.1 Package uppaal

A Uppaal System is represented as an object of the class **UppaalSystem**. It contains references to the processes (objects of the class **Instance**), the global variables and the requirements of this model. An **Instance** has two lists for locations and edges (represented by the classes **Location** and **Edge**). A Location consists of its invariant, the local variables and the information whether or not it is urgent or committed. In addition to that, each location has a unique ID. An edge contains references to its start and target locations as well as the select, guard and update expressions. Edges that synchronize on a channel are represented by the subclass **SynchEdge** of **Edge**. Expressions are represented as abstract syntax trees.
To obtain such a representation from an XML- and a .q-file, the class **UppaalParser** is used. Parsing is performed in the following steps.

Parsing Global Declarations

1. The global declarations are first transformed into an abstract syntax tree. The corresponding parser and the classes of the syntax tree were created using JJTree/JavaCC. To traverse the tree, JJTree supports the visitor pattern ([GHJV95]) by adding a `jjtaccept()` to each node class of the tree.

2. For each type declaration, an object of the corresponding subclass of **Type** is created. This type is then stored in a **TypeTable**. A **TypeTable** maps the names of the declared types to the corresponding object. It can also contain a reference to a parent-**TypeTable** (this is later used when parsing local declarations).

3. For each variable declaration, a corresponding variable object is created (if necessary, the type is looked up in the **TypeTable** from step 2). It can either be a subtype of **SimpleVariable** (i.e. **IntVariable**, **BoundedIntVariable**, **ClockVariable**, **ScalarVariable**, **BoolVariable**, **Channel**) or a **CompositeVariable** (for arrays and structs). Each composite variable contains a list of corresponding simple variables. If the declaration has an initializer, the corresponding expression is evaluated using the class **ExpressionEvaluator** and the value is added as initial value to the variable. The variables are stored in a **VarTable**.

Parsing Templates

For each declared template, a **Template** object is created. It contains the syntax trees of the parameter declarations and the local variable declarations and the XML-elements of the edges and locations.

Parsing the System Declaration

In this step, the templates are instantiated according to the declarations in the system definition (i.e. corresponding **Instance** object are created). To instantiate a template, the following steps are performed.

1. A new **Instance** object is created and a new **VarTable** and **TypeTable** with the global tables as parents are added.

2. The paramaters are processed:

    - for call-by-reference parameters, an entry with the name of the parameter and the passed variable is added to the local VarTable

  - for call-by-value parameters, the value of the passed expression is evaluated and then a new variable with this value as initial value is added to the local VarTable

3. The local type and variable declarations are processed.

4. The locations and edges of this template are instantiated.

Instantiating Locations and Edges

When locations or edges are instantiated, first the corresponding objects are created. Then the abstract syntax trees of the invariants or guards and update expressions are initialized. This means that the nodes representing variable expressions are decorated with the corresponding variable objects which are obtained from the local VarTable.

Parsing Requirements

Finally, the requirement declarations are parsed. The variable nodes of the syntax tree are, as discussed above, decorated with the corresponding variables. Furthermore, expressions of the form `process.location`, which are used to test if a process is in a certain location, are decorated with the corresponding processes and locations.

5.2 Package tcs

The package tcs is used to model transition constraint systems. It contains the following classes.

- Transition constraint systems are represented by an object of the class **TCS**. It contains lists of assertions for the initial condition, the error condition and the transition relation. If the *autoUpdate* option is enabled, for all variables v that are not explicitly excluded by `removeFromAutoUpdate(TCSVariable v)`, a constraint of the form $v' = v$ is automatically added to all transitions that don't contain an expression using v'.

- An **Assertion** consists of the values for the old and new *pc* and a list of constraints. The method `getGuard(TCSVarMap)` returns a new Assertion that contains only the constraints that don't use a primed variable. In contrast to that, the method `getUpdate(TCSVarMap)` returns only the constraints that do use primed variables.

- A **Constraint** consists of two simple expression together with a relational operator (either $\leq, \leq, =, \geq$ or $\geq$).
- A **SimpleExpression** can either be a constant expression, a variable or an arithmetic expression.
- A variable in a transition constraint system is represented using the class **TCSVariable**. It can either be a primed or unprimed variable. If it is a primed variable, it contains a reference to the corresponding unprimed variable and vice versa.
- The **TCSVarMap** is used to store the variables used in a TCS and to assign unique names to them. Usually the name stored in the variable object is used, but in case there is already another variable with that name, a new name is computed. This ensures that different variable objects always get different names (which will be useful when converting local variables of Uppaal systems).
- **TCSWriter** is used to output a TCS in the ARMC/Slab format.

5.3 Package Converter

This package is used to convert Uppaal system to transition constraint systems. The class **TCSVars** maps Uppaal variables to corresponding TCSVariables and stores the special TCSVariables (e.g. the pc variables and the substate variable). In addition to that, it stores the substate names that have been used so far and has a method that automatically find a new substate name.
The actual conversion is carried out by the class **Converter** using the approach developed in chapter 4. Before expressions are converted, they are first simplified according to the rules in Figure 5.1. They are then first transformed into negation normal form and then into disjunctive normal form.

Original expression	**Simplified expression**
e_1 `imply` e_2	`(not` e_1`) or` e_2
`forall(i:int[a,b])` e	$e[a/i]$ `and` $e[(a+1)/i]$ `and ... and` $e[b/i]$ where $e[x/i]$ means that all occurences of i in e are replaced by x
`exists(i:int[a,b])` e	$e[a/i]$ `or` $e[(a+1)/i]$ `or ... or` $e[b/i]$
e_1 `!=` e_2	`(`e_1 `<` e_2`) or (`e_1 `>` e_2`)`

Figure 5.1: Rules to simplify expressions

6 Evaluation

The correctness of the implementation has been validated on a number of examples. In this chapter, three of them are presented and analyzed.

6.1 Fischer's protocol

As a first example, we consider Fischer's protocol for mutual exclusion which is contained as a demo file in the Uppaal distribution and described in [BDL04]. It consists of a template `P` with parameter `pid`, as shown in Figure 6.1. The goal of the protocol is to ensure that only one process at a time can be in the critical section (i.e. the location `cs`). The system consists of a number of instances of `P` (with different values for `pid`).
Appendix B shows the generated code for two instances and the requirement `A[] not (P1.cs && P2.cs)`. The number of transitions for this rather small model is 34, which is larger than the number of states of the product automaton (16). However, if the number of instances is increased, the number of transition grows roughly linearly, as the table in Figure 6.2 indicates. Figure 6.3 correlates this to the number of states of the product automaton.

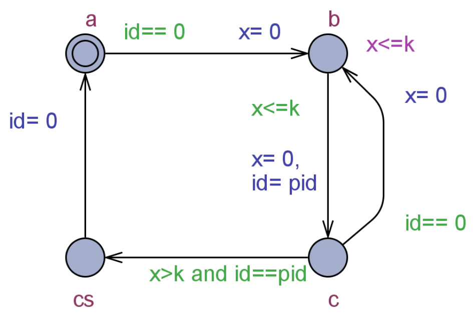


Figure 6.1: Template P of the Uppaal model for Fischer's protocol

Instances	Transitions
2	34
3	52
4	71
5	91
6	112
7	134
8	157

Figure 6.2: Number of transitions in the conversion of Fischer's Protocol

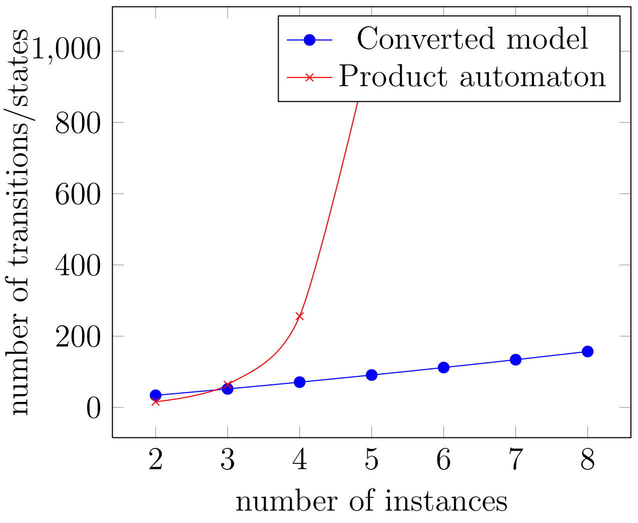


Figure 6.3: Relation between the number of transitions in the converted model and the number of states of the product automaton

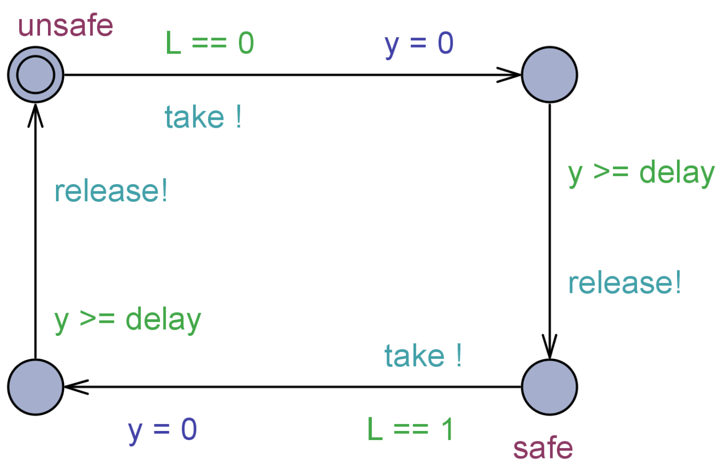


Figure 6.4: Template for vikings; the paramater delay denotes the time needed to cross the bridge

6.2 Bridge

This example is also part of the Uppaal distribution. It models the following scenario: Four vikings are about to cross a bridge in the middle of the night. The bridge is however damaged and can only carry two of the vikings at a time. Additionally, the vikings have only one torch and they cross the bridge with different speeds.
The system consists of a global clock `time`, two channels for taking and releasing the torch and a bounded integer variable `int[0,1] L` that denotes the side on which the torch currently is. The vikings are modeled by a template (Figure 6.4) using a parameter `const int delay` to represent the time needed to cross the bridge. Figure 6.5 models the torch. It has an urgent location to ensure that if two vikings cross the bridge together, they start at the same time.
The converted model has 72 transitions (compared to 1024 states of the product automaton).

6.3 Single-tracked Line Segment

This example is taken from [Mct]. It models “a distributed real-time controller for a segment of tracks where trams share a piece of track”. It is a fairly large model consisting of five processes with four to eight locations. The product automaton has 4608 states, the conversion obtained by our tool

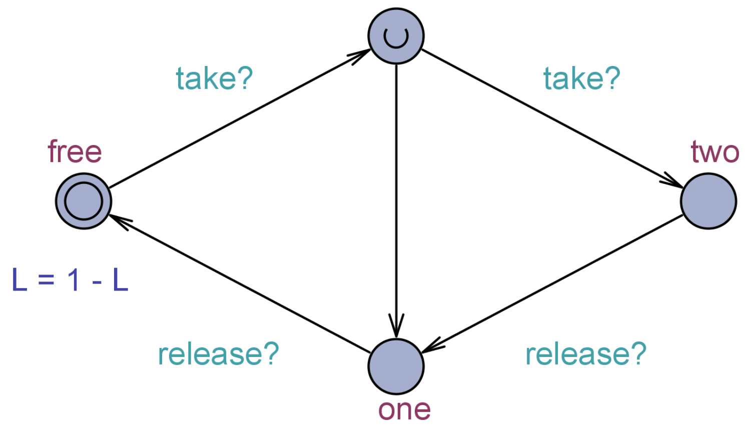


Figure 6.5: The automaton modeling a torch in the Bridge problem

has 439 transitions.

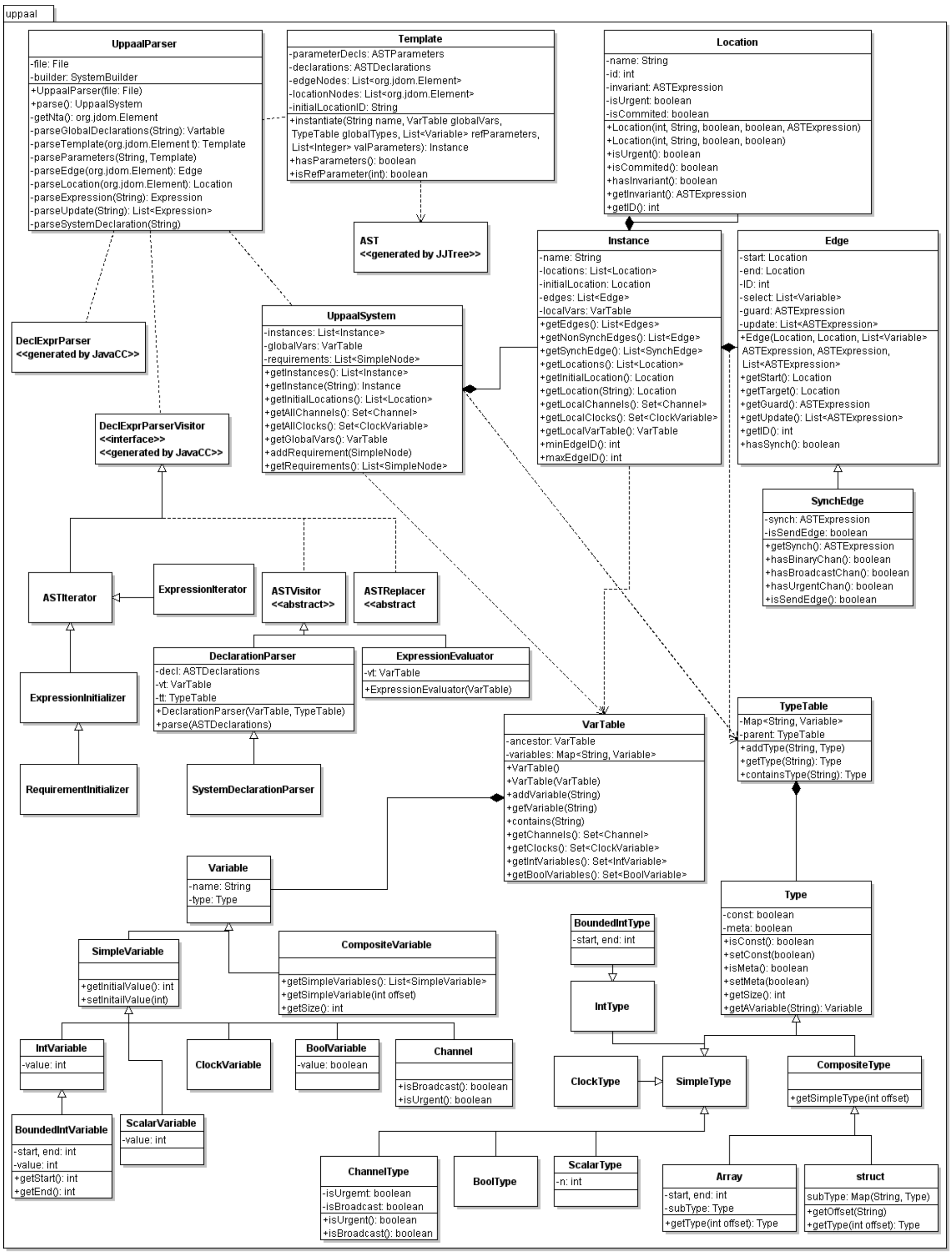
7 Conclusions

In this thesis, a tool that translates Uppaal systems into transition constraint system has been developed. First, the details of both approaches were introduced. In chapter 4 a mathematical description of our conversion method was given; the most important steps were proved formally. The method avoid creating the product automaton. It does so by partitioning one step of the timed automata network into several steps of the transition constraint system and by introducing for each automaton a new variable to denote its current location. The drawback of this technique is the introduction of additional deadlock states.
The tool developed in chapter 5 supports templates, binary synchronization, integer variables, typedefs, urgent and committed locations. Experiments on a number of different examples gave evidence that the tool works correctly and that the number of transitions is usually much smaller than the number of states of the product automaton.

7.1 Future Work

The tool could be enhanced by supporting more of the Uppaal specific features. For functions for instance this is however nontrivial and for features like arrays with non-constant expressions as indices this would most likely lead to a large blow-up in the number of transitions.
Apart from that, it might make sense to analyze how Slab/ARMC processes the converted files in order to find the parts on which most time is spent. This could be used to optimize the conversion method .

A
Class Diagrams



uppaal
UppaalParser
-file: File
-builder: SystemBuilder
+UppaalParser(file: File)
+parse(): UppaalSystem
-getNta(): org.jdom.Element
-parseGlobalDeclarations(String): Vartable
-parseTemplate(org.jdom.Element t): Template
-parseParameters(String, Template)
-parseEdge(org.jdom.Element): Edge
-parseLocation(org.jdom.Element): Location
-parseExpression(String): Expression
-parseUpdate(String): List<Expression>
-parseSystemDeclaration(String)
Template
-parameterDecls: ASTParameters
-declarations: ASTDeclarations
-edgeNodes: List<org.jdom.Element>
-locationNodes: List<org.jdom.Element>
-initialLocationID: String
+instantiate(String name, VarTable globalVars, TypeTable globalTypes, List<Variable> refParameters, List<Integer> valParameters): Instance
+hasParameters(): boolean
+isRefParameter(int): boolean
Location
-name: String
-id: int
-invariant: ASTExpression
-isUrgent: boolean
-isCommited: boolean
+Location(int, String, boolean, boolean, ASTExpression)
+Location(int, String, boolean, boolean)
+isUrgent(): boolean
+isCommited(): boolean
+hasInvariant(): boolean
+getInvariant(): ASTExpression
+getID(): int
AST
<<generated by JJTree>>
Instance
-name: String
-locations: List<Location>
-initialLocation: Location
-edges: List<Edge>
-localVars: VarTable
+getEdges(): List<Edges>
+getNonSynchEdges(): List<Edge>
+getSynchEdge(): List<SynchEdge>
+getLocations(): List<Location>
+getInitialLocation(): Location
+getLocation(String): Location
+getLocalChannels(): Set<Channel>
+getLocalClocks(): Set<ClockVariable>
+getLocalVarTable(): VarTable
+minEdgeID(): int
+maxEdgeID(): int
Edge
-start: Location
-end: Location
-ID: int
-select: List<Variable>
-guard: ASTExpression
-update: List<ASTExpression>
+Edge(Location, Location, List<Variable> ASTExpression, ASTExpression, List<ASTExpression>)
+getStart(): Location
+getTarget(): Location
+getGuard(): ASTExpression
+getUpdate(): List<ASTExpression>
+getID(): int
+hasSynch(): boolean
DeclExprParser
<<generated by JavaCC>>
UppaalSystem
-instances: List<Instance>
-globalVars: VarTable
-requirements: List<SimpleNode>
+getInstances(): List<Instance>
+getInstance(String): Instance
+getInitialLocations(): List<Location>
+getAllChannels(): Set<Channel>
+getAllClocks(): Set<ClockVariable>
+getGlobalVars(): VarTable
+addRequirement(SimpleNode)
+getRequirements(): List<SimpleNode>
DeclExprParserVisitor
<<interface>>
<<generated by JavaCC>>
SynchEdge
-synch: ASTExpression
-isSendEdge: boolean
+getSynch(): ASTExpression
+hasBinaryChan(): boolean
+hasBroadcastChan(): boolean
+hasUrgentChan(): boolean
+isSendEdge(): boolean
ASTIterator
ExpressionIterator
ASTVisitor
<<abstract>>
ASTReplacer
<<abstract
ExpressionInitializer
DeclarationParser
-decl: ASTDeclarations
-vt: VarTable
-tt: TypeTable
+DeclarationParser(VarTable, TypeTable)
+parse(ASTDeclarations)
ExpressionEvaluator
-vt: VarTable
+ExpressionEvaluator(VarTable)
TypeTable
-Map<String, Variable>
-parent: TypeTable
+addType(String, Type)
+getType(String): Type
+containsType(String): Type
VarTable
-ancestor: VarTable
-variables: Map<String, Variable>
+VarTable()
+VarTable(VarTable)
+addVariable(String)
+getVariable(String)
+contains(String)
+getChannels(): Set<Channel>
+getClocks(): Set<ClockVariable>
+getIntVariables(): Set<IntVariable>
+getBoolVariables(): Set<BoolVariable>
RequirementInitializer
SystemDeclarationParser
Variable
-name: String
-type: Type
Type
-const: boolean
-meta: boolean
+isConst(): boolean
+setConst(boolean)
+isMeta(): boolean
+setMeta(boolean)
+getSize(): int
+getAVariable(String): Variable
BoundedIntType
-start, end: int
SimpleVariable
+getInitialValue(): int
+setInitailValue(int)
CompositeVariable
+getSimpleVariables(): List<SimpleVariable>
+getSimpleVariable(int offset)
+getSize(): int
IntType
IntVariable
-value: int
ClockVariable
BoolVariable
-value: boolean
Channel
+isBroadcast(): boolean
+isUrgent(): boolean
ClockType
SimpleType
CompositeType
+getSimpleType(int offset)
BoundedIntVariable
-start, end: int
-value: int
+getStart(): int
+getEnd(): int
ScalarVariable
-value: int
ChannelType
-isUrgemt: boolean
-isBroadcast: boolean
+isUrgent(): boolean
+isBroadcast(): boolean
BoolType
ScalarType
-n: int
Array
-start, end: int
-subType: Type
+getType(int offset): Type
struct
subType: Map(String, Type)
+getOffset(String)
+getType(int offset): Type

Figure A.1: Class diagram for package uppaal

B
Generated code for fischer.xml

```
% Preamble:
:- multifile r/5,implicit_updates/0,var2names/2,preds/2,cube_size/1,start/1,error/1,
 refinement/1.
refinement(inter).
cube_size(1).
start(pc(init)).
error(pc(error)).

preds(p(_, data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)), []).

var2names(p(_, data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),
   [(_urgent, 'urgent'),
    (_len, 'len'),
    (_k, 'k'),
    (_pcP2, 'pcP2'),
    (_pid_0, 'pid'),
    (_clock_x_0, 'clock_x'),
    (_committed, 'committed'),
    (_id, 'id'),
    (_e_s, 'e_s'),
    (_pcP1, 'pcP1'),
    (_clock_x, 'clock_x'),
    (_e_r, 'e_r'),
    (_k_0, 'k'),
    (_pid, 'pid')]).

integral(_urgent, _k, _pcP2, _pid_0, _committed, _id, _e_s, _pcP1, _e_r, _k_0, _pid).
nonint(_len, _clock_x_0, _clock_x).

r(p(pc(init),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(default),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
 _clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[], [_idP=0,
 _pcP1P=2, _k_0P=2, _pid_0P=1, _pcP2P=6, _pidP=2, _kP=2, _lenP>=0, _clock_xP=_lenP,
 _clock_x_0P=_lenP, _e_rP=0, _e_sP=0, _urgentP=0],1).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent>0, 0*_committed=
 _committed, _id=0, _pcP1=2], [_lenP=0, _clock_x_0P=0+_lenP, _pcP1P=1, _urgentP=0+_urgent
 , _committedP=0+_committed, _clock_xP=_clock_x+_lenP, _kP=_k, _pcP2P=_pcP2, _pid_0P=
 _pid_0, _idP=_id, _e_sP=_e_s, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],2).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent=0, _id=0, _pcP1
 =2], [_lenP>=0, _clock_x_0P=0+_lenP, _pcP1P=1, _urgentP=0+_urgent, _committedP=0+
 _committed, _clock_xP=_clock_x+_lenP, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _idP=_id,
 _e_sP=_e_s, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],3).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(update_0),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
 _clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent>0, 0*
 _committed=_committed, _clock_x_0=<_k_0, _pcP1=1], [_lenP=0, _pcP1P=0, _urgentP=0+
 _urgent, _committedP=0+_committed, _clock_xP=_clock_x+_lenP, _kP=_k, _pcP2P=_pcP2,
 _pid_0P=_pid_0, _clock_x_0P=_clock_x_0, _idP=_id, _e_sP=_e_s, _e_rP=_e_r, _k_0P=_k_0,
 _pidP=_pid],4).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(update_0),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
 _clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent=0,
 _clock_x_0=<_k_0, _pcP1=1], [_lenP>=0, _pcP1P=0, _urgentP=0+_urgent, _committedP=0+
 _committed, _clock_xP=_clock_x+_lenP, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _clock_x_0P=
 _clock_x_0, _idP=_id, _e_sP=_e_s, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],5).
```

```
r(p(pc(update_0),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(update_1),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
 _clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[], [_clock_x_0P
=0+_lenP, _len=_lenP, _urgentP=_urgent, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0,
 _committedP=_committed, _idP=_id, _e_sP=_e_s, _pcP1P=_pcP1, _clock_xP=_clock_x, _e_rP=
 _e_r, _k_0P=_k_0, _pidP=_pid],6).
r(p(pc(update_1),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[], [_idP=_pid_0, _len=
 _lenP, _urgentP=_urgent, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _clock_x_0P=_clock_x_0,
 _committedP=_committed, _e_sP=_e_s, _pcP1P=_pcP1, _clock_xP=_clock_x, _e_rP=_e_r, _k_0P=
 _k_0, _pidP=_pid],7).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent>0, 0*_committed=
 _committed, _id=0, _pcP1=0], [_lenP=0, _clock_x_0P=0+_lenP, _pcP1P=1, _urgentP=0+_urgent
 , _committedP=0+_committed, _clock_xP=_clock_x+_lenP, _kP=_k, _pcP2P=_pcP2, _pid_0P=
 _pid_0, _idP=_id, _e_sP=_e_s, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],8).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent=0, _id=0, _pcP1
 =0], [_lenP>=0, _clock_x_0P=0+_lenP, _pcP1P=1, _urgentP=0+_urgent, _committedP=0+
 _committed, _clock_xP=_clock_x+_lenP, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _idP=_id,
 _e_sP=_e_s, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],9).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent>0, 0*_committed=
 _committed, _id=_pid_0, _clock_x_0>_k_0, _pcP1=0], [_lenP=0, _pcP1P=3, _urgentP=0+
 _urgent, _committedP=0+_committed, _clock_xP=_clock_x+_lenP, _clock_x_0P=_clock_x_0+
 _lenP, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _idP=_id, _e_sP=_e_s, _e_rP=_e_r, _k_0P=
 _k_0, _pidP=_pid],10).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent=0, _id=_pid_0,
 _clock_x_0>_k_0, _pcP1=0], [_lenP>=0, _pcP1P=3, _urgentP=0+_urgent, _committedP=0+
 _committed, _clock_xP=_clock_x+_lenP, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pcP2P=_pcP2
 , _pid_0P=_pid_0, _idP=_id, _e_sP=_e_s, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],11).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent>0, 0*_committed=
 _committed, _pcP1=3], [_lenP=0, _idP=0, _pcP1P=2, _urgentP=0+_urgent, _committedP=0+
 _committed, _clock_xP=_clock_x+_lenP, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pcP2P=_pcP2
 , _pid_0P=_pid_0, _e_sP=_e_s, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],12).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent=0, _pcP1=3], [
 _lenP>=0, _idP=0, _pcP1P=2, _urgentP=0+_urgent, _committedP=0+_committed, _clock_xP=
 _clock_x+_lenP, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0,
 _e_sP=_e_s, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],13).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent>0, 0*_committed=
 _committed, _id=0, _pcP2=6], [_lenP=0, _clock_xP=0+_lenP, _pcP2P=5, _urgentP=0+_urgent,
 _committedP=0+_committed, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pid_0P=_pid_0, _idP=_id
 , _e_sP=_e_s, _pcP1P=_pcP1, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],14).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent=0, _id=0, _pcP2
 =6], [_lenP>=0, _clock_xP=0+_lenP, _pcP2P=5, _urgentP=0+_urgent, _committedP=0+
 _committed, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pid_0P=_pid_0, _idP=_id, _e_sP=_e_s,
 _pcP1P=_pcP1, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],15).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(update_2),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
 _clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent>0, 0*
 _committed=_committed, _clock_x=<_k, _pcP2=5], [_lenP=0, _pcP2P=4, _urgentP=0+_urgent,
 _committedP=0+_committed, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pid_0P=_pid_0, _idP=_id
 , _e_sP=_e_s, _pcP1P=_pcP1, _clock_xP=_clock_x, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],16).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(update_2),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
 _clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent=0,
 _clock_x=<_k, _pcP2=5], [_lenP>=0, _pcP2P=4, _urgentP=0+_urgent, _committedP=0+
 _committed, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pid_0P=_pid_0, _idP=_id, _e_sP=_e_s,
 _pcP1P=_pcP1, _clock_xP=_clock_x, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],17).
r(p(pc(update_2),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(update_3),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
 _clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[], [_clock_xP
=0+_lenP, _len=_lenP, _urgentP=_urgent, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0,
 _clock_x_0P=_clock_x_0, _committedP=_committed, _idP=_id, _e_sP=_e_s, _pcP1P=_pcP1,
 _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],18).
r(p(pc(update_3),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
 _clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
 ,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[], [_idP=_pid, _len=_lenP,
  _urgentP=_urgent, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _clock_x_0P=_clock_x_0,
_committedP=_committed, _e_sP=_e_s, _pcP1P=_pcP1, _clock_xP=_clock_x, _e_rP=_e_r, _k_0P=
_k_0, _pidP=_pid],19).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent>0, 0*_committed=
_committed, _id=0, _pcP2=4], [_lenP=0, _clock_xP=0+_lenP, _pcP2P=5, _urgentP=0+_urgent,
_committedP=0+_committed, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pid_0P=_pid_0, _idP=_id
, _e_sP=_e_s, _pcP1P=_pcP1, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],20).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent=0, _id=0, _pcP2
=4], [_lenP>=0, _clock_xP=0+_lenP, _pcP2P=5, _urgentP=0+_urgent, _committedP=0+
_committed, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pid_0P=_pid_0, _idP=_id, _e_sP=_e_s,
_pcP1P=_pcP1, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],21).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent>0, 0*_committed=
_committed, _id=_pid, _clock_x>_k, _pcP2=4], [_lenP=0, _pcP2P=7, _urgentP=0+_urgent,
_committedP=0+_committed, _clock_xP=_clock_x+_lenP, _clock_x_0P=_clock_x_0+_lenP, _kP=_k
, _pid_0P=_pid_0, _idP=_id, _e_sP=_e_s, _pcP1P=_pcP1, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid
],22).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent=0, _id=_pid,
_clock_x>_k, _pcP2=4], [_lenP>=0, _pcP2P=7, _urgentP=0+_urgent, _committedP=0+_committed
, _clock_xP=_clock_x+_lenP, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pid_0P=_pid_0, _idP=
_id, _e_sP=_e_s, _pcP1P=_pcP1, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],23).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent>0, 0*_committed=
_committed, _pcP2=7], [_lenP=0, _idP=0, _pcP2P=6, _urgentP=0+_urgent, _committedP=0+
_committed, _clock_xP=_clock_x+_lenP, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pid_0P=
_pid_0, _e_sP=_e_s, _pcP1P=_pcP1, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],24).
r(p(pc(default),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(chInv),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,_clock_x_0P
,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[0+_urgent=0, _pcP2=7], [
_lenP>=0, _idP=0, _pcP2P=6, _urgentP=0+_urgent, _committedP=0+_committed, _clock_xP=
_clock_x+_lenP, _clock_x_0P=_clock_x_0+_lenP, _kP=_k, _pid_0P=_pid_0, _e_sP=_e_s, _pcP1P
=_pcP1, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],25).
r(p(pc(chInv),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(chInvP2),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
_clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[_pcP1=2], [
_urgentP=_urgent, _lenP=_len, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _clock_x_0P=
_clock_x_0, _committedP=_committed, _idP=_id, _e_sP=_e_s, _pcP1P=_pcP1, _clock_xP=
_clock_x, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],26).
r(p(pc(chInv),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(chInvP2),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
_clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[_clock_x_0=<
_k_0, _pcP1=1], [_urgentP=_urgent, _lenP=_len, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0,
_clock_x_0P=_clock_x_0, _committedP=_committed, _idP=_id, _e_sP=_e_s, _pcP1P=_pcP1,
_clock_xP=_clock_x, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],27).
r(p(pc(chInv),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(chInvP2),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
_clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[_pcP1=0], [
_urgentP=_urgent, _lenP=_len, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _clock_x_0P=
_clock_x_0, _committedP=_committed, _idP=_id, _e_sP=_e_s, _pcP1P=_pcP1, _clock_xP=
_clock_x, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],28).
r(p(pc(chInv),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(chInvP2),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
_clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[_pcP1=3], [
_urgentP=_urgent, _lenP=_len, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _clock_x_0P=
_clock_x_0, _committedP=_committed, _idP=_id, _e_sP=_e_s, _pcP1P=_pcP1, _clock_xP=
_clock_x, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],29).
r(p(pc(chInvP2),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(default),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
_clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[_pcP2=6], [
_urgentP=_urgent, _lenP=_len, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _clock_x_0P=
_clock_x_0, _committedP=_committed, _idP=_id, _e_sP=_e_s, _pcP1P=_pcP1, _clock_xP=
_clock_x, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],30).
r(p(pc(chInvP2),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(default),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
_clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[_clock_x=<_k,
_pcP2=5], [_urgentP=_urgent, _lenP=_len, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0,
_clock_x_0P=_clock_x_0, _committedP=_committed, _idP=_id, _e_sP=_e_s, _pcP1P=_pcP1,
_clock_xP=_clock_x, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],31).
r(p(pc(chInvP2),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(default),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
_clock_x_0P,_committedP,_idP,_e_sP,_pcP1P,_clock_xP,_e_rP,_k_0P,_pidP)),[_pcP2=4], [
_urgentP=_urgent, _lenP=_len, _kP=_k, _pcP2P=_pcP2, _pid_0P=_pid_0, _clock_x_0P=
_clock_x_0, _committedP=_committed, _idP=_id, _e_sP=_e_s, _pcP1P=_pcP1, _clock_xP=
_clock_x, _e_rP=_e_r, _k_0P=_k_0, _pidP=_pid],32).
r(p(pc(chInvP2),data(_urgent,_len,_k,_pcP2,_pid_0,_clock_x_0,_committed,_id,_e_s,_pcP1,
_clock_x,_e_r,_k_0,_pid)),p(pc(default),data(_urgentP,_lenP,_kP,_pcP2P,_pid_0P,
_clock_x_0P ,_committedP ,_idP ,_e_sP ,_pcP1P ,_clock_xP ,_e_rP ,_k_0P ,_pidP)) ,[_pcP2=7], [
_urgentP=_urgent , _lenP=_len , _kP=_k , _pcP2P=_pcP2 , _pid_0P=_pid_0 , _clock_x_0P=
_clock_x_0 , _committedP=_committed , _idP=_id , _e_sP=_e_s , _pcP1P=_pcP1 , _clock_xP=
_clock_x , _e_rP=_e_r , _k_0P=_k_0 , _pidP=_pid],33).
r(p(pc(default),data(_urgent ,_len ,_k ,_pcP2 ,_pid_0 ,_clock_x_0 ,_committed ,_id ,_e_s ,_pcP1 ,
_clock_x ,_e_r ,_k_0 ,_pid)),p(pc(error),data(_urgentP ,_lenP ,_kP ,_pcP2P ,_pid_0P ,_clock_x_0P
,_committedP ,_idP ,_e_sP ,_pcP1P ,_clock_xP ,_e_rP ,_k_0P ,_pidP)) ,[_pcP2=7, _pcP1=3], [],34).
```

Bibliography

[AD90] Rajeev Alur and D. L. Dill. Automata for modeling real-time systems. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 322–335, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[BDFW07] Ingo Brückner, Klaus Dräger, Bernd Finkbeiner, and Heike Wehrheim. Slicing abstractions. In Farhad Arbab and Marjan Sirjani, editors, *Proceedings of the International Symposium on Fundamentals of Software Engineering (FSEN)*, 2007.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. 2004. www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. pages 87–124. 2004. http://www.cs.aau.dk/ adavid/UDBM/materials/by04-bookchapter.pdf.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[Had09] Walid Haddad. Verifying networks of phase event automata. Master's thesis, Saarland University, 2009. `http://react.cs.uni-sb.de/index.php?id=517`.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1992.

[Hoe06] Jochen Hoenicke. *Combination of Processes, Data, and Time.* PhD thesis, University of Oldenburg, July 2006. http://csd.Informatik.Uni-Oldenburg.DE/ skript/pub/Papers/csp-oz-dc.pdf.

[Mct] Mcta. `http://mcta.informatik.uni-freiburg.de/benchmarks.html`.

[RP07] Andrey Rybalchenko and Andreas Podelski. ARMC: The logical choice for software model checking with abstraction refinement. In

PADL, volume 4354 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2007. http://www.mpi-sws.org/ rybal/papers/padl07-armc.pdf.