REDUCING UPPAAL MODELS THROUGH CONTROL FLOW ANALYSIS

G.H. Slomp

COMPUTER SCIENCE
FORMAL METHODS & TOOLS

EXAMINATION COMMITTEE
prof.dr. J.C. van de Pol
dr.ir. R. Langerak
dr. M. Weber

UNIVERSITY OF TWENTE.

24-09-2010

Abstract

This paper present a dead variable analysis algorithm for reducing the state space of UPPAAL models. By resetting irrelevant variables to their initial value reductions of UPPAAL models are achieved.
The developed algorithm consists of two parts. In the first part we define an algorithm to determine the relevance of variables. We also cope with the various features of UPPAAL like, for instance, function calls and value passing variables and we present an algorithm to determine the relevance of variables that are used in a property. In the second part we transform the original timed automata by introducing resets for irrelevant variables. We improve on extending transformation algorithms by not resetting irrelevant variables at every location.
Based on the developed algorithm we implemented a tool that performs the transformation and by executing this tool for three case studies we show that it indeed achieves reductions.
We conclude with noting that, next to the reductions, the main benefit of our work is that UPPAAL users do not have to perform the resets manually any more, making their work easier and less error prone, as these resets can now be performed automatically.

Contents

CHAPTER 1

Introduction

Model checking has its roots in the 1970's, when Computer Science was facing the problem of Concurrent Program Verification [10]. Errors in concurrent programs were hard to reproduce due to the concurrency in the programs. Model checking solved this problem because it offered the possibility to exhaustively search all possible execution paths within a program. A simple definition of model checking is:

"Given a model M and a formula φ, model checking is the problem of verifying whether or not φ is true in M (written $M \models \varphi$)."

One way to represent a system as a model is by using automata, which have the advantage that besides the formal definition they can also be represented graphically. For instance, look at the automaton in figure 1.1, modelling the behaviour of a simple coffee machine. The automaton represent a coffee machine that delivers coffee after pressing the button once, but delivers cappuccino after pressing the button twice.

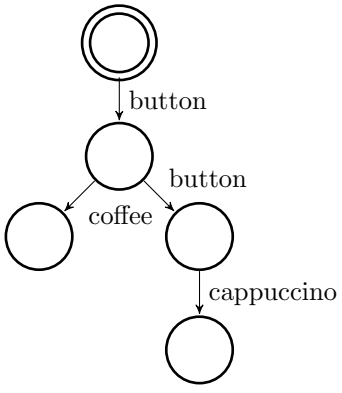


Figure 1.1: Automaton of a simple coffee machine

Having this model of figure 1.1 we want to know if the corresponding coffee machine is working properly and that it is, for example, guaranteed that you will get a cup of cappuccino if you press the button twice and do not end up with a cup of coffee. This property can be expressed in computational tree logic (CTL) as: button.button $\Rightarrow$ cappuccino.

The property is satisfied and we could say that the machine is working correctly. But what if somebody presses the button and an hour later someone else presses the button. In practice, after the first time the button is pressed a cup of coffee is produced and the counting starts again, but in theory this does not hold. What we actually would want is that we can say something about the time that has progressed after the button is pressed for the first time.

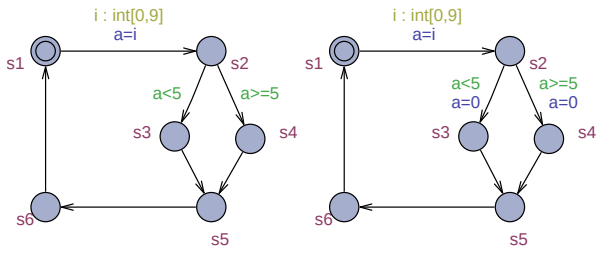
This is exactly what researchers encountered when they wanted to check more complex systems as for real-time systems the standard automata were not sufficient because a notion of time was required. Therefore, to model the behaviour of real time systems, Alur and Dill extended standard automata with clocks, timing constraints and clock resets, resulting in *timed automata* [2]. Based on these timed automata various tools were developed, which can be used in the verification of real-time systems. Two well-known examples are Kronos [13] and UPPAAL [25]. In this research we will focus on UPPAAL, a tool developed by Uppsala University, Sweden and Aalborg University, Denmark. We have chosen for UPPAAL as it is the most-used tool for modelling real timed systems.

Originally UPPAAL was based on timed automata only, but since the newer versions, it uses a combined approach, also supporting imperative code. This gives users the possibility to define, for instance, functions and call these functions from the automata [4]. It has been used in various real-life problems. For instance, an error in an audio/video protocol by Bang & Olufsen has been found and the corrected protocol has (partially) been proven correct [17]. Another example is the analysis of the Philips Audio Control Protocol using UPPAAL [6]. Both case studies are relatively large and in both cases it takes UPPAAL several minutes to generate an error trace or prove the modelled protocol correct.

Increasing the efficiency of proving such (and other) models would result in faster verification but it also enables the verification of larger models. Therefore it is beneficial to reduce the state space. For this there are several possible options like, for example, reduce the memory consumption, space consumption or the number of states that are stored. Recent research by Van de Pol and Timmer [30] shows a method to achieve the reduction in size of the state space. Their research focusses on an intermediate format of process algebraic specifications, linear process equations (LPEs). Their algorithm is an excellent candidate to achieve similar results in the area of timed automata. Therefore we will adapt their approach to UPPAAL specifications to achieve a similar result.

1.1 Initial exploration

As mentioned, research by van de Pol and Timmer [30] suggests that it is worth trying to adapt and apply their method on models defined in UPPAAL. The following will give an introductory view of possible reductions.



(a) Original model (b) Optimised model

Figure 1.2: Basic example showing the possible reductions (data variables)

File	States explored	States stored
Before	50	50
After	15	15

Table 1.1: Results of the verification

In UPPAAL there are two types of variables which both seem to be suitable for control flow reconstruction in order to reduce the state space. (Discrete) data variables are the first type of variables, both global and local. Figure 1.2(a) shows a relatively simple model, which selects a value of zero to nine and assigns the value to a variable a (see section 2 for a introduction to the UPPAAL language or section 3 for a formal description). This variable a is then used in the (guard of the) next edge, but, after that, is not used any more till it is assigned a random value again. The value of a is only relevant in state $s2$, while in the other states it is only a cause of a growth in the number of states stored. By resetting the value of a in the transition from $s2$ to $s3$ or $s4$, see figure 1.2(b), a reduction in the state space can be achieved. To test the reduction we check a simple property $A\Box x \geq 0$,which is always true, in order to compare the number of states explored/stored. Using no space optimisation we get the results as shown in table 1.1 If we take a look at the second type of variable, clock variables, the first impression is that there are less possibilities to reduce the state space. The reason for this is the fact that the clocks are stored using zones [8]. For a data variable x, which can have values ranging from 0–9, for every location there can be up to 10 different states just varying in the value of x. However, a clock variable c will be stored in a state using a zone $0 \leq c \leq 9$ and therefore resetting its value does not automatically reduce the number of states. Even the model of figure 1.3 results in only 4 states, where you would expect more than 4 states in the generated state space. However this

is not the case as UPPAAL only generates 4 states, where you would expect different states for the 4 zones ; $c \geq 0, c \geq 1, c \geq 2$, resulting in 16 states. The conclusion is that for clock variables we probably cannot reduce the state space that much (due to the clock zones), but for data variables there are certainly some reductions possible.

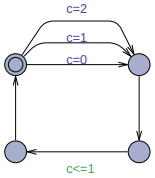


Figure 1.3: Model with a state space of only four states

1.2 Research Questions

After the initial exploration we now define the research question of the project: 'What can control flow analysis applied to UPPAAL models achieve?'. To answer this question we answer the following sub questions:

- Can the algorithm of [30] be translated to reduce the state space of UPPAAL models by resetting local variables?
- Is there a way to reduce the state space by resetting global variables without constructing the total state space?
- How can the algorithm be extended to include the state invariants of UPPAAL?
- How can we incorporate the 'different' features of UPPAAL into the algorithm?
- Is the tool beneficial for end users of UPPAAL, releasing them from optimizing the model for efficient verification.

CHAPTER 2

Related work

For a good understanding of the following sections a basic knowledge of (timed) automata is required. We give a complete formal description in section 3, but for now a short informal description is sufficient. In figure 2.1 a simple on/off switch is modelled. The circles represent the states/nodes of the automaton and the arrows between states represent the edges or transitions of the system. The automaton of figure 2.1 starts in the *OFF* state (indicated by the double circle) and can go to the *ON* state by processing a *Push* action. Once in the *ON* state another *Push* action moves the automaton back to the *OFF* state.

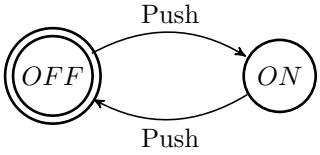


Figure 2.1: Automaton representing an on/off switch

A simple extension of the above are the *guards*. A guard is a boolean expression that 'guards' the transition and needs to evaluate to *true* for the transition to be enabled. If a guard evaluates to $false$ the transition is not enabled and cannot be taken. If we add a boolean variable *broken* (initially $false$) to the automaton of figure 2.1 and a guard $!broken$ to the transition $OFF \xrightarrow{Push} ON$ we get the automaton of figure 2.2, which does not allow the transition to be taken if the switch is broken.

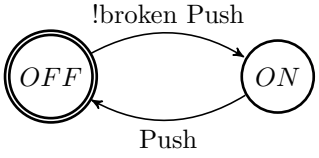


Figure 2.2: Automaton representing an on/off switch with a guard

A great variety of systems can be modelled with the above automata, but how do we model for instance a dimmer which can be turned on by pushing the button once and can be put in dimmed state by pushing the button again in at most 5 seconds? For this problem (and other problems) a notion of time has been added to automata to get *timed automata* [2]. In figure 2.3 the dimmer is presented as an automaton using a real-valued clock x which can be reset to 0. If the automaton receives a first $Push$ the clock x is reset to measure the passed time since the $Push$. If another $Push$ occurs within 5 seconds the transition to DIM will be taken, otherwise the dimmer is turned OFF.

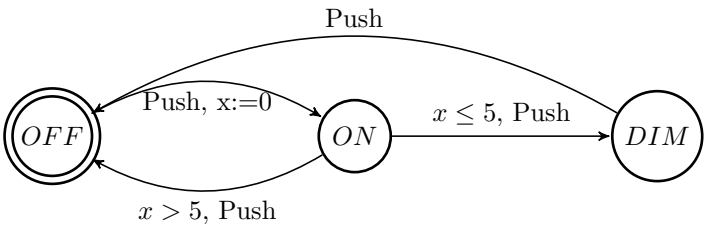


Figure 2.3: Automaton representing a dimmer

In section 1 we mentioned UPPAAL, a model checker for real-timed systems, which can be used to model and verify these timed automata. We also noted that UPPAAL, like model checkers in general, suffers from the state space explosion problem. There are various approaches to solve this problem, which we present in the following sections. We first give an overview in section 2.1 of various techniques to reduce the size of the generated state space, followed by a description of some compiler optimisation techniques.

2.1 Reduction techniques

The following sections give an overview of the various reduction techniques that are available. We conclude with mentioning for each technique if its available in UPPAAL and automatically enabled.

2.1.1 Symmetry Reduction

Symmetry reduction is one of the techniques to reduce the number of states to be explored and is available in UPPAAL since UPPAAL 4.0 [19, 4]. It applies the idea of symmetry reduction of Ip and Dill [21] to UPPAAL and uses the occurrence of multiple identical processes only differing in the their identity, also called full symmetry [21]. By defining an equivalence group, based on an automorphism, large reductions in the verification process can be achieved. To implement symmetry reduction two problems should be solved, the problem of detecting the automorphism from the system description and the problem of deciding the symmetry of two states during verification. Therefore the data type *scalarset* was added, which is a sub-range with restricted operations and is a fully symmetric type, resulting in the behaviour of the program being invariant

under permutation of the elements of the scalar-set. For scalar-sets of size n reductions of up to a factor $n!$ can be achieved.

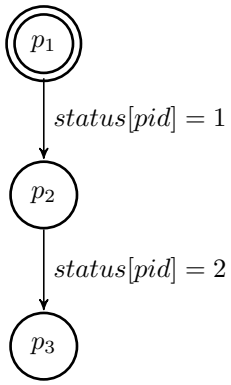


Figure 2.4: Automaton of a process P keeping track of his status

For example consider the process in figure 2.4, which only keeps track of the current status of the process using an array called status. In the first transition the process sets his status to 1 using the variable *pid* as index and in the next transition the status is set to 2. If we have multiple of these processes, for instance two, we will get the state space of figure 2.5, which is made of all possible interleavings of the two processes. A non-dashed edge is a transition of the first process, while a dashed edge is a transition of the second process. Notice the complete symmetry of the automaton. Symmetry reduction makes use of the symmetry by exploring only a part of the total state space, which is the set of filled nodes in figure 2.5. Observe that all the non-filled nodes have an corresponding filled node only differing in (the order of) their pid's. For example $(0, 2) \equiv (2, 0)$, or in general $(i, j) \equiv (j, i)$.

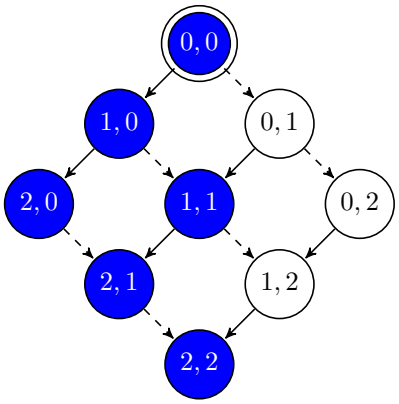


Figure 2.5: The composition of two instances of the process of figure 2.4

2.1.2 Slicing

Another method is that of *slicing*, an abstraction technique. Abstraction techniques use a part, an abstraction, of the model, in order to verify a property for the whole model. Because only a part of the model is verified the state space that needs to be generated is much smaller. However one has to make sure the abstraction preserves the properties that are verified, otherwise a correct abstraction does not guarantee the correctness of the whole model. In [22] a first approach is presented using slicing with timed automata, while in [29] it is showed how slicing can be used in the current version of UPPAAL, which uses not solely timed automata any more but is extended with new data types and user defined functions. Slicing reduces the original model to a set of relevant components with regards to some slicing criteria. These criteria are based on the locations and variables of the property to be verified. Figure 2.6 gives an example of an abstraction slicing produces. We define a property '$\forall\Box$ not deadlock', which guarantees us that the process will never deadlock. If we verify this property the statement 'nrOfRuns++' is irrelevant as it does not affect the control flow or another variable but serves as a status variable. The slicing algorithm will remove this statement from the specification and the result is that in the total state space the size of each state vector is smaller.

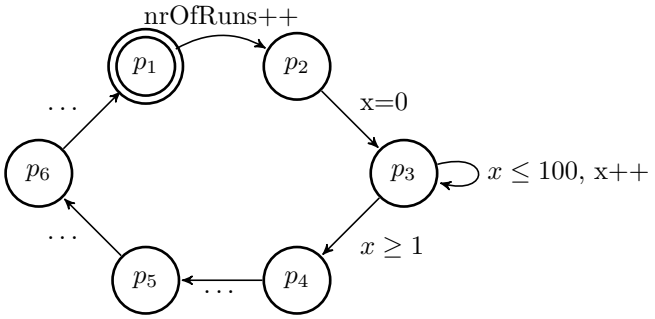


Figure 2.6: Example explaining the slicing algorithm

2.1.3 Partial order reduction

A well known method to reduce the state space is *partial order reduction*. Normally the next state to be explored is chosen from $enabled(s)$, all transitions that are enabled/possible in the state s. Partial order reduction tries to use a set $ample(s) \subseteq enabled(s)$ instead [26]. If one can define a set $ample(s)$ smaller than $enabled(s)$ the resulting state space will be smaller. The set $ample(s)$ is generated by looking at the the interleaving of independent edges (transitions). Two edges α and β are independent if:

- $\alpha \in enabled(\beta(s))$ and $\beta \in enabled(\alpha(s))$ - They do not disable each other
- $\alpha(\beta(s)) = \beta(\alpha(s))$ - The order in which they are executed does not matter.

Because these edges are independent it does not matter in which order they are traversed and as a consequence it is beneficial to only traverse one possible interleaving of those edges. In the area of timed automata partial order reduction is a little harder because there the interleaving of, for instance, two clock resets leads to two different states and does not produce the nice diamond structure known in partial order reduction. Looking at figure 2.7 one has 2 processes with one clock each. If, in the combined automata, x is reset first and in the next transition y one ends up in a state (r_3) where $x \geq y$ holds whereas the other way around one ends in a state (r_5) where $x \leq y$ holds, whereas, in the case of data variables states r_3 and r_5 would be the same. To apply partial order reduction to timed automata the idea of letting the local clocks proceed independently of the clocks of other processes is presented [7]. This means that whenever a synchronised action is performed the local clocks still need to be synchronised and therefore extra clocks are added to each process. A prototype has been implemented but this implementation does not show large reductions, mostly because of the introduction of a large number of extra local clocks [3].

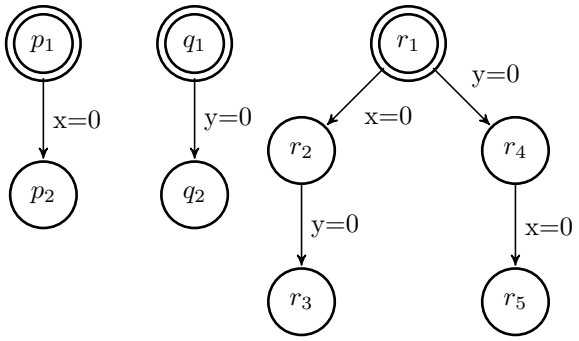


Figure 2.7: Interleaving two clock resets

2.1.4 Dead variable analysis

In figure 2.8 a simple automaton is presented. In this automaton a random value (ranging from 0 to 9) is assigned to x and in the next transition this value x is added to the value of y. In the next two transitions the variable x is not used and then the loop starts again and x is assigned a new random value. Because the value of x is not used in p_3 and p_4 and there does not exist a path to another state where x is used, we say that x is not relevant in p_3 and p_4. The other way around we can see that x is relevant in p_2 as x is used in a outgoing transition from p_2. We can say x is relevant in p_3. To reduce the state space we could reset a variable if it is not relevant. This analysis of relevant variables is called dead (or live) variable analysis.
Several papers have presented an algorithm for state space reduction based on dead/live variable analysis. For instance [9] (only sequential processes) and [32], however both algorithms only check the relevance of a variable locally. If a variable is used globally at multiple processes it is automatically relevant. For instance if a variable is passed to another process it is automatically relevant even if it not used in the other process. This is because, in the case of parallel

processes, the algorithm first tries to reduce the processes separately, without looking at the specification of a parallel process, before composing the combination of all the parallel processes. Also [**?**] and [30] (for LPEs) present work on dead variable analysis and even the tutorial on UPPAAL [5] references dead variable analysis and gives modelling tips how to manually apply the analysis to reduce the state space. However the analysis is not integrated into UPPAAL and therefore not automatically performed.

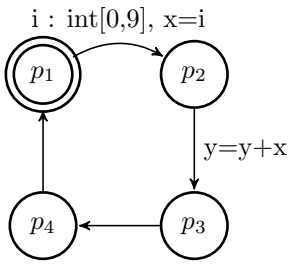


Figure 2.8: Trivial example to demonstrate dead variable analysis

2.1.5 Exact (clock) acceleration

Hendriks and Larsen address the problem of unnecessary fragmentation of the state space, due to different time scales in the real time system [18]. This occurs, for example, when a systems samples the environment many times each second, whereas the environment only changes a couple of times each second. In a automata this can be seen as a cycle which can only be left if a clock reaches a certain value, however in the meantime the cycle is repeated many times, resulting in a very large state space because non of these iterations of the cycle overlap. They propose an algorithm, called *exact acceleration*, to transform this cycle such that there is a new cycle which will only be traversed once resulting in a much smaller state space.

2.1.6 Active clock reduction

One of the reasons of the state space explosion is the large number of clocks. In [14] two reasons are given for the large number of clocks. The first is the fact that most of the time specifications are written in a higher level language and translated to timed automata replacing each time-out with a own clock. Most of the time, however, these clocks are not used, or active, at the same time and the number clocks could be reduced. Secondly most of the time a timed automaton is made of a large number of smaller components, each with its own clocks. It turns out that some of these clocks can be replaced by one clock, because these clocks are reset at the same time. Active clock reduction [14] is an algorithm, implemented in UPPAAL, that makes use of the above to reduce the number of states and the size of the states in the state space.

2.1.7 Reduction techniques in UPPAAL

In table 2.1 we indicate for each of the reduction techniques if it is implemented in UPPAAL and if it's automatically enabled in UPPAAL. Below the table we give some remarks on the entries of the table.

Technique	**In UPPAAL?**	**Automatically enabled?**
Symmetry Reduction	+	+- (1)
Slicing	- (2)	-
Partial order reduction	- (2)	-
Dead variable analysis	- (3)	-
Exact clock acceleration	- (2)	-
Active clock reduction	+	+

Table 2.1: Overview of reduction techniques

1. For symmetry reduction you have to annotate an integer range as a scalar set if it is fully symmetric. If you do this the verification automatically applies symmetry reduction.
2. For all of these techniques tools/prototypes have been implemented, but there is not a complete working implementation available in UPPAAL of these techniques.
3. One of the results of this paper is a tool that can perform dead variable analysis for UPPAAL models. It is not implemented in UPPAAL but is available as an preprocessing tool.

2.2 Compiler optimisation techniques

All the methods described in the previous section are all about optimising the model checking process. However there are other research areas that have optimisation techniques. Those techniques may be interesting for our project as they can prove useful. One interesting area is the area of compiler optimisation. Almost every compiler nowadays makes use of the Single Static Assignment form which we present first, followed by various compiler optimisation techniques of which some may prove useful for our project.

2.2.1 Static Single Assignment form

Static single assignment (SSA) [11] form is not an optimisation itself, however it is an special representation of the original code that makes it possible that other algorithms/techniques, that do cause optimisations, can be easily applied. Basically a program is in SSA form if each variable is a target of exactly one assignment statement. Using SSA form it is easier to see which variable assignment corresponds to a particular use of a variable. Consider the following piece of code:

$$y := 1$$
$$y := 2$$
$$x := y$$

It is easy to see that the first assignment has no use as the value it assigns will never be used, but one can imagine this will be harder when the code gets more complex. Even for this small piece of code it is clear that SSA form makes it easier to draw conclusions about the use of variables. To translate the piece of code we assign to every variable an unique (for that variable) index number. Each use of a variable will get the same index number as the corresponding definition of that variable. For the example this results in:

$$\begin{aligned} y_1 &:= 1 \\ y_2 &:= 2 \\ x_1 &:= y_2 \end{aligned}$$

With the piece of code in SSA form we can now directly conclude that y_1 is never used (and therefore that the assignment is useless). However the transformation is not as trivial as it seems, for instance what to do when there are conditional branches like in the following piece of code:

$$\begin{aligned} y \quad &:= 3; \\ if \quad &(z > 5)\{ \\ &\quad y := y + 3; \\ \} \\ x \quad &:= y; \end{aligned}$$

Now it is not clear which indexed version of y needs to be assigned to x. Therefore a special function, called a ϕ-function is inserted, which will take care of this decision for us. This results in the following code in SSA-form:

$$\begin{aligned} y_1 \quad &:= 3; \\ if \quad &(z_1 > 5)\{ \\ &\quad y_2 := y_1 + 3; \\ \} \\ y_3 \quad &:= \phi(y_1, y_2); \\ x_1 \quad &:= y_3; \end{aligned}$$

The exact implementation of the ϕ-function is not important, however its result is that the correct assignment is chosen based on which branch/control flow is taken. One of the difficult steps in transforming to SSA form is to determine where to exactly place these ϕ-functions.

Dominance Frontiers

In [11] an efficient algorithm is presented to determine the placement of ϕ functions. The algorithm uses *dominance frontiers* to calculate the placements. The algorithm uses 2 relations between control flow nodes, *dominates* and *strictly dominates*. They are defined as follows:

Let X and Y be two nodes of the control flow graph and *Entry* the starting point of the control flow graph.

- X dominates Y ($X \gtrdot Y$) if X appears on every path from *Entry* to Y.
- X strictly dominates Y ($X \gg Y$) if X dominates Y and $X \neq Y$.

Based on these definitions of dominance dominance frontiers (DF) of a node, the point where dominance of a node stops and and another control flow path joins the current path (this indicates a possible ambiguity about which definition to use), can be defined as:

$$\mathrm{DF}(X) = \{Y | \exists P \in \ \mathrm{Pred}(Y))(X \gtrsim P \text{ and } X \neg \gg Y)\}$$

Control Dependency Graph

Also [11] shows us that dominance frontiers can be used to determine control dependences in the control flow graph. The reverse control flow graph is exactly the same as the control flow graph but with every edge $X \rightarrow Y$ being replaced by an edge $X \leftarrow Y$. The same algorithm to determine the *dominance frontiers* can now be used to determine the control dependency's. Every node is control dependent on the nodes in its dominance frontier in the reversed control flow graph.

2.2.2 Dead Code Elimination

The term *dead code* can have several meanings. Some people define dead code as unreachable code, whereas other people define it as ineffectual code. In [11] an algorithm is presented that eliminates ineffectual code using a control dependency graph. The algorithm itself is easy to understand and comes down to the following:

Initially all statements are marked *dead* and a statement is marked live if:

- The statement affects program output (I/O, reference parameter assignment or routine call with side effects)
- Assignment statement
  - Its outputs already used in a live statement
- Conditional branch and a live statement is control dependent on it.

We could use *dead code implementation* in our project to eliminate ineffectual assignments, making sure that any irrelevant variable has a standard value and the resulting state space is minimal. However we do not eliminate statements at the moment in our project but only reset irrelevant variables. (See for instance the future work described in section 8.1)

2.2.3 Code Motion

Simply said *code motion* is the movement of statements to optimize program execution. There are several versions of code motion algorithms, for instance an implementation oriented algorithm for lazy code motion [23], which tries to minimize the number of computations while suppressing any unnecessary code motion. Another algorithm is presented by Cytron et al. in [12], who claim that their algorithm is capable of performing code motion even if abstractions are used.

However all the algorithms have the same goal, that is to move statements to a better, more efficient, location. Consider the following example:

$$\begin{array}{ll} \text{for} & (i = 0; i < n; i{+}{+})\{ \\ & x = y + z; \\ & a[i] = 6 * i + x * x; \\ \} & \end{array}$$

For each iteration of the loop the value of x is computed again however the values of y and z do not change inside the loop, therefore the value of x receives the same value every time. It would be beneficially to move the computation of x outside the loop in order to avoid a redundant computation. The same applies to the computation of $x * x$. This produces the following:

$$\begin{array}{ll} x = y + z; & \\ t1 = x * x; & \\ \text{for} & (i = 0; i < n; i{+}{+})\{ \\ & a[i] = 6 * i + t1; \\ \} & \end{array}$$

The result of applying *code motion* to (a piece of) code is that (some) redundant computations are performed less. This results in an increased performance as the execution time will be reduced, however the state space will not be affected, making the algorithm not useful for our project.

2.2.4 Constant Propagation

Constant propagation [31] is a global control flow analysis problem and its goal is to identify values that are always constant and to propagate these values as far through the program as possible. Expressions with constant operands are again constant and this fact can, for instance, be used in further computations. In [31] several uses for compilers are given:

- If you can evaluate an expression at compile time you do not have to compute it every time during runtime, increasing the performance of a program.
- Unreachable code can be deleted. This can happen if a conditional branch is never taken because the value of the condition is constant.
- Since many of the calls to procedures are constant, using constant propagation together with procedure integration can have beneficial results.

Most of the above uses of constant propagation are of no use to our project. However, the fact that unreachable control branches can be detected could prove to be useful. It may cause variables to be marked irrelevant whereas they else would be unnecessarily marked relevant.

2.2.5 Global Value Numbering and redundant computations

In [27] an optimisation is presented that is based on the SSA form and makes use of *Global Value Numbering* to identify *redundant computations*. The algorithm

improves the program because it uses the fact that it is not efficient to perform the same computation again. A basic example is:

$$A := C$$
$$D := A * B$$
$$E := C * B$$

The above assignments contain some redundancy as D and E are assigned the same value. The algorithm will identify this and the second assignment can be deleted and replaced by an simpler assignment. These kind of transformations cause an increase in performance because redundant computations are performed less. However the state space size will be the same as before and therefore the algorithm is not useful for our project.

CHAPTER 3

UPPAAL

After briefly introducing UPPAAL in the section we will now take a closer look at UPPAAL by looking at the syntax and semantics of the UPPAAL language. UPPAAL is based on the definition of timed automata, which originates from the work of Alur and Dill [2]. They introduced timed automata as an extension to finite state automata and they added a notion of time, by introducing clocks, to the automata. This addition of time includes the possibility to add constraints over the clocks to the edges (called guards) and to the locations (called invariants). In the following sections we will give a description of the syntax and semantics of these timed automata and the additions made by UPPAAL. The work of Thrane and Sørensen [29] provides a thorough description of these syntax and semantics and therefore large parts of the following sections come directly from the work of Thrane and Sørensen.

3.1 Basic timed automata

We begin with the basic definitions of timed automata, which are mainly based on the work of Alur and Dill [2]. After the basic definitions we will extend the definitions with the extensions made by UPPAAL.

Definition 1 (**Timed Automaton**). A *timed automaton* is a tuple $\langle L, l^0, \Sigma, C, E, I\rangle$ where:

- L is a finite set of locations
- $l^0 \in L$ is the initial location
- Σ is a finite set of channels
- C is a finite set of clocks
- $E \subseteq L \times \Psi(C) \times \Sigma \times 2^C \times L$ is the set of edges
  - $\Psi(C)$ is the set of constraints over the set of clocks C (section 3.1.1)
- $I : L \longrightarrow \Psi(C)$ assigns each location with a set of invariants

We use $l \xrightarrow{g,a,r} l'$ to denote $\langle l, g, a, r, l'\rangle \in E$ where $l, l' \in L$ are locations (source and target respectively), $g \in \Psi(C)$ is the clock constraint guarding the edge, a is the channel, with $a = z! \mid z?|\ \epsilon$ and $z \in \Sigma$, which in some cases may be referred to as the action and $r \in 2^C$ is the set of clocks that are reset.

3.1.1 Clocks

Clocks are one of the most important features of timed automata as they are the feature that allow us to model real timed systems using automata. Clocks are initially zero and are increased synchronously at the same rate. We use C to denote the set of clocks in an automaton.

Clock valuations

A clock valuation is a total mapping $\sigma : C \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative real numbers. For $\delta \in \mathbb{R}_{\geq 0}$, $\sigma + \delta$ denotes an updated clock valuation σ', such that $\forall u \in C : \sigma'(u) = \sigma(u) + \delta$. The clock valuation σ_0 gives the initial valuation such that $\forall u \in C : \sigma_0(u) = 0$. Finally, $\mathscr{C}$ is used to denote the set of clock valuations.

Clock resets

Clock resets are used to reset the value of a clock variable to zero, its initial value. The result of resetting a set of clocks r is defined as $\sigma' = \sigma[r \mapsto 0]$, which means that for every clock $c \in r \subseteq C$ the value of c is set to 0, while the value of the other clocks remains unchanged.

Clock constraints

Constraints on clocks are used as guards on edges and invariants at locations. A constraint ψ in the set of clock constraints $\Psi(C)$, may be of the following form:

$$\psi, \psi_1, \psi_2 ::= u \sim n | u - u' \sim n | \psi_1 \wedge \psi_2$$

for $u, u' \in C, \sim \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$. Satisfiability of a clock constraint $\psi \in \Psi(C)$ by a clock valuation σ is defined inductively on the structure of ψ by

$$\begin{array}{lcl} \sigma \models u \sim n & \iff & \sigma(u) \sim n \\ \sigma \models u - u' \sim n & \iff & \sigma(u) - \sigma(u') \sim n \\ \sigma \models \psi_1 \wedge \psi_2 & \iff & \sigma \models \psi_1 \text{ and } \sigma \models \psi_2 \end{array}$$

3.1.2 Channels

A notion of channels is used to obtain synchronisation between timed automata in a network (in parallel). Edges of timed automata are decorated with channels from the alphabet Σ. We say that a timed automaton is willing to output, if it is able to take an edge which is decorated with $a!$ where $a \in \Sigma$. Alternatively, we say a timed automaton is willing to input if it is able to take an edge which is decorated with $a?$, where $a \in \Sigma$. Two timed automata, of a network P of timed automata, may synchronize whenever one is willing to output to some channel and the other is willing to input on the same channel from a global set of channels Σ_P. Channels can also be grouped into an array and to access a single channel of this array you have to use channel[expr].

3.1.3 Semantics

The semantics of timed automata are defined as a *timed labelled transition system* (TLTS) where states (or configurations) consist of a location $l \in L$ and a clock valuation $\sigma \in \mathscr{C}$. Transitions are either delay transitions, denoted by $\xrightarrow{d}$, with $d \in \mathbb{R}_{\geq 0}$, or they are action transitions, denoted by $\xrightarrow{a}$, with $a \in \Sigma$. A system may either delay in the current location, while the location's invariant stays satisfied, or follow an outgoing, enabled edge (i.e. an edge where the current clock valuation satisfies the guard) in the system decorated by channel a.
Because invariants and guards are defined as sets of predicates over clocks i.e. $\Psi(C)$, we use the notation $\sigma \models I(L)$ to mean that σ satisfies $I(L)$.

Definition 2 (**Semantics of Timed Automata**)**.** The semantics of timed automata is defined in terms of a TLTS where states are pairs $\langle l, \sigma \rangle$ of locations and clock valuations and the transitions are defined by the rules.

- $\langle l, \sigma \rangle \xrightarrow{d} \langle l, \sigma + d \rangle$ if $(\sigma + d') \models I(l)$ for all $d' \in \mathbb{R}_{\geq 0}$ where $d' \leq d$
- $\langle l, \sigma \rangle \xrightarrow{a} \langle l', \sigma' \rangle$ if $l \xrightarrow{g,a,r} l'$ s.t. $\sigma \models g \wedge \sigma' = \sigma[r \mapsto 0] \wedge \sigma' \models I(l')$

3.1.4 Parallel composition

We use the term network to denote a model of parallel composed timed automata. A network of timed automata P is defined over a common set of clocks and channels and consists of n timed automata $P_i = \{L_i, l_i^0, C, \Sigma, E_i, I_i\}$, where $1 \leq i \leq n$. A location in P is a *location vector* $\bar{l} = \{l_1, \ldots, l_n\}$ over locations for each P_i. Updates to the location vector are written $\bar{l}[l_i'/l_i]$ to denote that automaton P_i moves from location l_i to l_i'. We proceed to define the semantics of networks of timed automata. We use the invariant function $I(\bar{l})$ to denote the conjunction of terms from $I_i(l_i)$.

Definition 3 (**Timed Automata Networks**)**.** Let $P = \langle L_i, l_i^0, C, \Sigma, E_i, I_i \rangle$ be a parallel composition of timed automata $(P_1 \parallel \ldots \parallel P_n)$ and let $\langle \bar{l}, \sigma \rangle$ be an element in the set of states $S = (L_1 \times \ldots \times L_n) \times \mathscr{C}$ where $s_0 = (\bar{l^0}, \sigma_0)$ denotes the initial state where $\bar{l}^0 = (l_1^0, \ldots, l_n^0)$. The semantics is defined in terms of a timed labelled transition system $\langle S, s_0, \rightarrow \rangle$ and the transition relation $\rightarrow \subseteq S \times S$ is defined by:

- $\langle \bar{l}, \sigma \rangle \xrightarrow{d} \langle \bar{l}, \sigma + d \rangle$ if $\forall d' : \sigma + d' \models I(\bar{l})$ where $0 \leq d' \leq d$
- $\langle \bar{l}, \sigma \rangle \xrightarrow{a} \langle \bar{l}[l_i'/l_i], \sigma' \rangle$ if $\exists l_i \xrightarrow{g,a,r} l_i'$ s.t. $\sigma \models g, \sigma' = \sigma[r \mapsto 0]$ and $\sigma' \models I(\bar{l}[l_i'/l_i])$
- $\langle \bar{l}, \sigma \rangle \xrightarrow{\tau} \langle \bar{l}[l_i'/l_i, l_j'/l_j], s' \rangle$ if there exists $l_i \xrightarrow{g_i,a!,r_i} l_i'$ and $l_j \xrightarrow{g_j,a?,r_j} l_j'$ s.t. $i \neq j$, $\sigma \models (g_i \wedge g_j)$, $\sigma' = \sigma[r_i \cup r_j \mapsto 0]$ and $\sigma' \models I(\bar{l}[l_i'/l_i, l_j'/l_j])$

3.2 The Extended Timed Automata Formalism

The notion of extended timed automata introduced here is based on the previous definitions. The 'upgrades' are primarily concerned with the addition of discrete variables and replacing the previously introduced resets by updates expressed in a small imperative language. The core motivation behind these extensions, is

that the formalism, obviously, may now be used to model a richer set of systems where not only time is of importance but also the value of discrete data.

Variable Valuation

In order to extend the definition of timed automata with discrete variables, we introduce a notion of variable valuations. A variable valuation is a total mapping $\omega : V \to \mathbb{Z}$ from a set of variables V to the set of integers. The variable $retVal$ is a special variable which is solely used for returning values from function calls. Finally, we use $\mathscr{V}$ to denote the set of all variable valuations.

3.2.1 Syntax of the Imperative Language

This section introduces a subset of the imperative language of UPPAAL, which we will use throughout this thesis. The language presented here is chosen such that the correctness of our reduction algorithm introduced in later chapters can be argued not only to hold for this subset, but it could be extended to the full imperative language of UPPAAL. For instance a for-loop can be easily transformed into a while-loop with the same semantics. Therefore we will not introduce both constructs but only show the while-loop.

Functions and statements In order to manipulate discrete variables, we introduce the possibility of having functions, which may be called, for instance, in the update part of the edges. We use f to denote a function and F to denote a set of functions defined in the following syntax (*bexpr* will be introduced in the next paragraph).

funcDecl	::=	type f(id$_1$,...,id$_n$){stmt_seq}
stmt_seq	::=	ϵ | single_stmt stmt_seq
single_stmt	::=	if(bexpr){stmt_seq} else{stmt_seq}
		| while(bexpr){stmt_seq}
		| return expr | single_act| single_asg

Here we make a slight extension to the syntax as explained in the work of Thrane and Sørensen [29] by adding **single_asg** to **single_stmt** and we add the if-else statement instead of the if-statement. The first change is because we do not make a difference between data and clock variables, therefore there is no need to separate both definitions. Secondly, the expressiveness of if-else-statements is greater than the expressiveness of if-statements. In addition every if-statement can be written as if-else-statement like this: if φ stmt_seq else skip.
In the above syntax **type** is the type of the function and specifies the type of the return value, which can be either int or bool, but also records or arrays of those two. Secondly **id** is used to denote names of formal parameters i.e. locally declared discrete variables. As is traditional for imperative languages, the body of functions or the branching and looping constructs are composed of a, possibly empty, sequence of statements given by the production **stmt_seq**. The syntax for function calls is defined as:

funCall	::=	f(expr$_1$,...,expr$_n$),where $f \in F$

Expressions. Let V be a finite set of integer variables. The arithmetic expression over V, using the set of functions F , is defined in the following grammar as **expr**, where $m \in \mathbb{Z}, v \in V$ and $\otimes \in \{-, +, *, /\}$.

$$\text{expr} \quad ::= \quad m \,|\, v \,|\, \text{expr} \otimes \text{expr} \,|\, -\text{expr} \,|\, \text{funCall}$$

By $Expr(V, F)$, we denote the set of all possible arithmetic expressions over V and F.
The set of boolean expressions over discrete variables is defined in the production **bexp**, where **expr** $\in Expr(\mathcal{V}, F)$ and $\sim \in \{==, =, <, >, <=, >=\}$.

$$\text{bexp} \quad ::= \quad \text{true} \mid \text{expr} \sim \text{expr} \,|\, \text{bexp} \;\&\&\; \text{bexp} \,|\, \text{bexp} \| \text{bexp} \,|\, \neg \text{bexp}$$

The set of all boolean expressions over V and F is denoted by $\Phi(V, F)$ ranged over by φ.
Finally, actions over discrete variables V and functions F are defined by the production **single_act** , where $v \in V$ and **expr** $\in Expr(V, F)$. The set of all actions over V and F is denoted by $Act(V, F)$.

$$\text{single\_act} \quad ::= \quad \text{funCall} \,|\, v = \text{expr} \,|\, \text{skip}$$

Clocks. Let C be a finite set of real valued variables, called clocks. The set of clock constraints over C is defined in the production `clockconst`, where $u, u_1, u_2 \in C, c \in \mathbb{N}$ and $\sim$ is defined as before.

$$\text{clockconst} \quad ::= \quad \text{true} \;|\, u \sim c \,|\, u_1 - u_2 \sim c \,|\, \text{clockconst} \&\& \text{clockconst}$$

By $\Psi(C)$ we denote the set of all clock constraints over C, ranged over by ψ. As with discrete variables, we use a production **single_asg** to define clock assignments, where $Asg(C, F)$, denotes the set of all assignments over C.

$$\text{single\_asg} \quad ::= \quad u = \text{expr} \;|\, \text{skip}$$

Not all assignments are possible as assignments to clocks are limited to the regular = assignment operator and only integer expressions are allowed on the right hand side of such assignments.

Additional restrictions on the syntax by UPPAAL

In addition to the syntax presented in the previous section UPPAAL has some restrictions on what is allowed and what is not. Clearly from the syntax it is possible that a single_stmt expands to a single_act and this single_act into function call. However UPPAAL does not allow recursive calls as each function call has to be preceded by its accompanying function declaration. This also ensures that functions A and B cannot enter an infinite loop where they keep calling each other.

The other statements of UPPAAL

The syntax of UPPAAL as presented here does not contain the full syntax of UPPAAL. Next to the if- and while-statements there are various other statements that are allowed in UPPAAL but to keep things clear they are not mentioned here. We give an overview of these options and show that they are captured by the other statements and therefore, effectively, introduce no other functionality. During the rest of this thesis we therefore ignore these statements (at least in the theoretical part, of course they will be implemented).

Do-while The syntax of UPPAAL for a do-while statement is the following:

$$\text{do } \{\text{stmt\_seq}\} \text{ while } (\varphi)$$

If we compare this to the while statement which is 'while(φ){stmt_seq}' one can see that they are almost similair with the only difference that for a do-while statement the condition is not evaluated prior to the first execution of the stmt_seq. We can easily rewrite this to a while-statement and then we get the following:

$$\text{stmt\_seq while}(\varphi)\{\text{stmt\_seq}\}$$

For-loop UPPAAL has 2 versions of a for-loop. The first is java/c++ like and looks like this:

$$\text{for}(\ \text{expr}_{init}\ ;\ \varphi\ ;\ \text{expr}_{incr})\ \{\text{stmt\_seq}\}$$

Also this statement can be transformed into a while-statement and the resulting list of statements is:

$$\text{expr}_{init}\ ;\ \text{while}(\varphi)\{\text{stmt\_seq}\ ;\ \text{expr}_{incr}\}$$

The other for-loop version can only be used in conjunction with a scalar set and executes the accompanying stmt_seq for each element of the scalar set. One can see that this can be transformed into a list of stmt_seq one for each element of the scalar set.

Switch/Case Next to the above statements the C++ library of UPPAAL, UTAP (see chapter 6), also includes the switch/case and default statements, suggesting that it is possible to use these statements in an UPPAAL model. However the help file does not mention how to use this statement and manually trying to use it in a model does not seem to work either at the moment. Therefore we do not consider this statement, but if it is necessary one can transform the switch statement using multiple if/else-statements.

3.2.2 Syntax of Extended Timed Automata

Having introduced the imperative syntax used in the notion of the extended time automata formalism, we proceed to extend the definition of the timed automata:

Updates: The notion of resets in the original definition has been replaced by updates. An update is a sequence of variable actions and clock assignments . As we will not make a distinction between clock and data variables in our reduction algorithm we combine both the definitions of variable and clock assignments (single_act and single_asg) resulting in the following syntax.

update ::= ϵ | update update | single_act | single_asg

Every update part of an edge can contain a clock assignment, a data variable assignment, a function call or nothing at all (skip). This set of all update actions over C,V and F is *Update(V,C,F)*.

Guards and invariants We extend the previously defined notion of guards and invariants with the discrete data and the use of the imperative language. Both guards and invariants are conjunctions over clock constraints and discrete boolean expressions denoted ψ and φ respectively. In addition, we restrict the valuation of discrete boolean expressions to be side-effect-free. Effectively reducing the semantics of guards and invariants to $\mathscr{C} \times \mathscr{V} \rightarrow \{true, false\}$.

Communication and urgency The extended timed automata have two other extensions. The first is the notion of broadcast, to enable synchronisation between multiple timed automata. The second extension is the ability to model the fact that time can not delay in a location, by marking locations committed or urgent. Roughly speaking locations can be marked *urgent*, time cannot delay at a location marked urgent. Even more restrictive is a location marked *committed*. In this case, if one or more of the locations of a state are marked commuted, not only time cannot delay, also the next transition should be from a location marked *committed*.
In the following definition we use $\eta(\Phi(V,F), \Psi(C))$ to denote the set of all conjunctions over $\Phi(V,F)$ and $\Psi(C)$.

Definition 4 (**The Extended Timed Automata**)**.** Let $P = \langle L, l^0, V, C, \Sigma, F, E, I\rangle$ be a timed automaton extended with discrete variables.

- L is a finite set of locations, ranged over by l
  - Each location is either marked urgent or committed or not marked at all.
- l^0 is the initial location
- V is a finite set of discrete variables, ranged over by v
- C is a finite set of clocks, ranged over by u
- Σ is the finite set of channels, ranged over by a
- F is a set of function declarations expressed in the above syntax
- $E \subseteq L \times \eta(\Phi(V,F), \Psi(C) \times \Sigma \times \mathit{Update}(V,C,F) \times L$ is the set of edges
- $I : L \rightarrow \eta(\Phi(\mathcal{V},F), \Psi(\mathcal{C}))$ assigns each location an invariant

3.2.3 Semantics of the imperative language

For a complete description of the semantics of the language of the extended timed automata we refer you to section 2.3.4 of the work of Thrane and Sørensen

[29]. We now give only the part that copes with the semantics of the transition relation:

The transition relation $\rightarrow \subseteq S \times S$ is defined by the following rules:

Let $\langle \bar{l}, \sigma, \omega \rangle$ be an element in the set of states $S = (L_1 \times \ldots \times L_n) \times \mathcal{C} \times \mathcal{V}$.

Delay: $\langle \bar{l}, \sigma, \omega \rangle \xrightarrow{d} \langle \bar{l}, \sigma + d, \omega \rangle$ if:

- $\forall d'$ where $0 \leq d' \leq d : (\sigma + d', \omega) \models I(\bar{l})$
- And $\forall d', l \in \bar{l} : d'$ does not result in an edge e being enabled for any l, which is either *urgent* or *committed*.

Action: $\langle \bar{l}, \sigma, \omega \rangle \xrightarrow{a} \langle \bar{l}[l'_i/l_i], \sigma', \omega' \rangle$ if:

- there exists an edge $l_i \xrightarrow{g,a,r} l'_i$ where
- $(\sigma, \omega) \models g$ and
- $a = \epsilon$
- $(\sigma', \omega') = [\![\text{update}]\!](\sigma, \omega)$
- $(\sigma', \omega') \models I(\bar{l}[l'_i/l_i])$
- And l_i is committed or there is no edge e that is enabled for any l_j such that l_j is committed

Although Thrane and Sørensen [29] describe the semantics of committed and urgent in the delay step they do not describe the semantics of committed in the action step. Therefore we have added the last item to the semantics of the action step.

Sync: $\langle \bar{l}, \sigma, \omega \rangle \xrightarrow{\tau} \langle \bar{l}[l'_i/l_i, l'_j/l_j], \sigma', \omega' \rangle$ if:

- there exist edges $l_i \xrightarrow{g_i, a!, r_i} l'_i$ and $l_j \xrightarrow{g_j, a?, r_j} l'_j$ and a state (σ'', ω'') such that
- $(\sigma, \omega) \models g_i \wedge g_j$
- (output) $(\sigma'', \omega'') = [\![\text{update}]\!](\sigma, \omega)$ (followed by)
- (input) $(\sigma', \omega') = [\![\text{update}]\!](\sigma'', \omega'')$.
- $(\sigma', \omega') \models I(\bar{l}[l'_i/l_i, l'_j/l_j])$
- And l_i and/or l_j are committed or there is no edge e that is enabled for any $l_k \in barl$ such that l_k is committed

Notice that UPPAAL does not require that the intermediate state satisfies the guard and the invariant. Formally, $(\sigma'', \omega'') \models g_j \wedge I[l'_i/l_i]$ does not have to be satisfied, because the communication is viewed as an atomic step.

CHAPTER 4

Relevance of variables

After having presented the syntax and semantics of UPPAAL in chapter 3 we present in this chapter the algorithm to determine the relevance of variables at locations. A variable is relevant if it influences the control flow of the program and/or the result of an property check. An irrelevant variable can be assigned any value without changing the control flow or the outcome of the verifier. We also give an equivalence relation based on this relevance of variables. While these two steps resemble in essence [30] there are several differences, such as the elimination of the function calls from the statements and the extended functionality of UPPAAL. The main complications are how to cope with arrays, value passing variables and property specifications. The final step, defining the transformation of the original model, will take place in chapter 5. For a simple explanation of the main idea look at the example presented in section 1.1. However, before we define the relevance algorithm we define some auxiliary terminology and a couple of rewriting rules to eliminate complex behaviour such as function calls.

4.1 Terminology

Recall from section 3.1 that we have $l \xrightarrow{g,a,r} l'$ to denote $\langle l, g, a, r, l' \rangle \in E$. We will write $src(e)$, $guard(e)$, $channel(e)$, $update(e)$, $target(e)$ to reference to the components, l, g, a, r, l' respectively, of an edge. The guards and invariants are expressed as constraints on both clock and data variables whereas $update(e)$ is a bit more complicated and can contain both clock and data variables, assignments using both types of variables and function calls. The functions are even more complicated allowing, for instance, control structures like *if* and *while*. Therefore we first look more closely at the update part of the edges.

4.1.1 Rewriting of the update statements

For a correct use of the relevance algorithm, which we introduce in section 4.2, we need to be able to reason about the code at statement level and also about the order of the various statements. However we have the problem that in UPPAAL there is a distinction between the statements that are allowed inside

functions (single_stat) and the statements that are allowed at the update part of an edge (single_asg or single_act), while for a simple, correct algorithm one would prefer to have both set of statements to be the same. Therefore we extend the set of statements allowed at the update part of an edge, resulting in both sets of statements being equal. In this way we do create a larger language, but, because the language of UPPAAL remains a subset of this language, we will not encounter any problems processing UPPAAL models.

First we recall the syntax of UPPAAL and then extend this to our extended language model. In UPPAAL, recall section 3.2.1, we have the following syntax:

funcDecl	::=	$f(\text{id}_1, \ldots, \text{id}_n)\{\text{stmt\_seq}\}$
$d$stmt_seq	::=	$\epsilon \mid$ single_stmt stmt_seq
single_stmt	::=	if (bexpr) {stmt_seq} else {stmt_seq}
		| while (bexpr) {stmt_seq} | return expr | single_act| single_asg
single_act	::=	funCall $\mid v =$ expr $\mid$ skip
single_asg	::=	$u =$ expr $\mid$ skip

By combining single_stmt with single_act and single_asg we get a new language model with no more distinction between statements inside functions and statements directly on an edge. Until now we have used u and v for respectively clock variables and data variables, however from this point on we use u and v to represent both kind of variables, as we shall not make a clear distinction between both types of variables.

funcDecl	::=	$f(\text{id}_1, \ldots, \text{id}_n)\{\text{stmt\_seq}\}$
stmt_seq	::=	$\epsilon \mid$ single_stmt stmt_seq
single_stmt	::=	if (bexpr) {stmt_seq} else {stmt_seq} | while (bexpr) {stmt_seq}
		| return expr | funCall $\mid u =$ expr | skip

To make it easier to to use above definitions in our algorithms we also eliminate the use of stmt_seq and replace it by a sequential composition of single_stmt.

funcDecl	::=	$f(\text{id}_1, \ldots, \text{id}_n)\{\text{single\_stmt}\}$
single_stmt	::=	if (bexpr) {single_smt} else{single_smt} | while (bexpr) {single_smt}
		| return expr | funCall $\mid u =$ expr | skip | single_stmt ; single_stmt

The update part of an edge then becomes:

update	::=	single_stmt

The next step in our design is the elimination of function calls by expanding the statements inside the function definition while keeping the same behaviour of the model. This can be done because function calls cannot be recursive, making sure that the number of function calls to unfold is not infinite. By removing the function calls we eliminate the 'unexpected' side-effects of functions. We also remove function calls and complex expressions as array indices, resulting in array indices containing either a constant value or a variable value consisting of just one variable, while the complex structure is concentrated in the assignment statements.

The only hard part of eliminating function calls is the fact that a function declaration can have multiple exit points. There are two possible solutions

to handle multiple exit points. The first is to assume we only have models with functions that have single exit points. The other solution is to adapt the algorithms of section 4.2 and/or chapter 5 in order to correctly pass on the relevance to a function declaration. Because the second option results in much more complex algorithms for the relevance and the transformation we choose the first.

How to rewrite a multi-exit function into a single-exit function? For a better understanding of both concepts and an idea about why multi-exit functions provide more difficulties we show a simple example. If we look at listing 4.1 we see a simple function which has a single return point at the end. Assume we call the function from an assignment to a relevant variable a, like $a = \text{sum}(b, c)$. Without considering every detail of the relevance algorithm which we present in section 4.2 first the return value becomes relevant because this return value is used in the rhs of an assignment to a relevant variable. Secondly, as we process the last statement the return value becomes not relevant and x and y become relevant instead.The final step is that are b and c are marked as relevant variables at the point of the function call, while x and y are marked not relevant. By this final step we also ensure that local variables do not become relevant outside their scope.

```
1 int sum(int x, int y){
2    return x+y;
3 }
```

Listing 4.1: Listing of a single-exit function

However, if we take a look at listing 4.2 we see that there are multiple return points. If we mark the return value relevant for this function and process the last statement the return value becomes not relevant again and instead y is marked relevant. At the time we process the other return statement, the return value is no longer relevant and x does not become relevant, which is not the result we would want.

To solve this we show in listing 4.3 how we can rewrite this function into a function with the same behaviour using an auxiliary variable and an else-branch. Another solution, as mentioned, would be to use a different relevance algorithm, however that will turn out to be more complex. Because it is possible to rewrite a multi-exit function into a single-exit function with the same behaviour assume, for simplicity, we only deal with single-exit functions.

```
1 int highest(int x, int y){
2    if(x>y){  return x;}
3    return y;
4 }
```

Listing 4.2: Listing of a multi-exit function

```
1 int highest(int x, int y){
2    if(x>y){  highest = x;}
3    else { highest = y ;}
4    return highest;
5 }
```

Listing 4.3: Listing of the multi-exit function rewritten to single-exit

Simple Timed Automata Before we define how we eliminate the function calls and the other complex structures we first introduce a special notion of timed automata that we use throughout the rest of this thesis. We call this a *simple timed automaton* and it is an extended timed automaton which complies to the following characteristics:

- No function calls
- Every condition check (if(v) or while(v)) contains only a single variable.
- Every array index (a[v]) contains only a single variable.
- Every expression is of the form $u_1 \otimes u_2$ with u_1 and u_2 variables
- All functions are single-exit functions.

In extension to the above differences between a simple timed automaton and an extended time automaton we also assume:

- Every variables identifier is used uniquely, assuring that there are not a global and a local variable with the same identifier.

Definition 5 (Rewriting and 'eliminating' function calls)**.** We define a function Λ that transform a statement (single_stmt) into a list of statements (single_stmt) thereby creating a simple timed automaton from a timed automaton. This function Λ is inductively defined by the following rules:

In all of the rules below we define v_i to be a fresh variable, a variable that is not used yet in the rest of the model. Also we use $\sim$ to represent every boolean operator (a combination of the previously used $\sim$ and $\otimes$)

1. $\Lambda(u = \ldots)$
   (a) $\Lambda(u = \text{CONSTANT}) = \{u = \text{CONSTANT}\}$
   (b) $\Lambda(u = \text{VAR}) = \{u = \text{VAR}\}$(integer, clock or boolean variable)
   (c) $\Lambda(u = f(\text{expr}_1, \ldots, \text{expr}_n)) = \Lambda(f(\text{expr}_1, \ldots, \text{expr}_n)) +\!\!+ \{u = \text{retVal}\}$
   (d) $\Lambda(u = a[\text{expr}_1][\ldots][\text{expr}_n]) = \Lambda(v_1 = \text{expr}_1, \ldots, v_n = \text{expr}_n) +\!\!+ \{u = a[v_1][\ldots][v_n]\}$
   (e) $\Lambda(u = \text{expr}_1 \sim \text{expr}_2) = \Lambda(v_1 = \text{expr}_1, v_2 = \text{expr}_2) +\!\!+ \{u = v_1 \sim v_2\}$
   (f) $\Lambda(u = \neg\text{expr}) = \Lambda(v_1 = \text{expr}) +\!\!+ \{u = \neg v_1\}$
2. $\Lambda(\text{return expr}) = \Lambda(\text{retVal} = \text{expr})$.
3. $\Lambda(\text{skip}) = \emptyset$
4. For *if* and *while* statements we have:
   - $\Lambda(\text{if}(\varphi)\{\text{stmt\_seq}\}) = \Lambda(v_1 = \varphi) +\!\!+ \{\text{if}(v_1)\{\Lambda(\text{stmt\_seq})\}\}$
   - $\Lambda(\text{while}(\varphi)\{\text{stmt\_seq}\}) = \Lambda(v_1 = \varphi) +\!\!+ \{\text{while}(v_1)\{\Lambda(\text{stmt\_seq}, v_1 = \varphi)\}\}$
   - $\Lambda(\text{if}(\varphi)\{\text{stmt\_seq}\}\ \text{else}\ \{\text{stmt\_seq}\}) = \Lambda(v_1 = \varphi) +\!\!+ \{\text{if}(v_1)\{\Lambda(\text{stmt\_seq})\}\ \text{else}\ \{\Lambda(\text{stmt\_seq})\}\}$
5. For a sequence of statements we have:

- $\Lambda(\lambda_1, \ldots, \lambda_n) = \Lambda(\lambda_1, \ldots, \lambda_{n-1}) +\!\!+ \Lambda(\lambda_n)$

6. For a function call we have:
   - $\Lambda(f(e_1, \ldots, e_n)) = \Lambda(p_1 = e_1, \ldots, p_n = e_n) +\!\!+ \Lambda(\lambda_1, \ldots, \lambda_n)$
     - if there exists a accompanying function declaration: $f(p_1, \ldots, p_n)\{\lambda_1, \ldots \lambda_n\}$

The definition above may seem quite complex and several design decisions may seem unclear at the moment. Therefore we give some examples and explain the decisions made.

Handling return expressions (1 and 2) Due to the fact that it is possible to have function calls in the right hand side of an assignment we have to consider the side-effects of this function call. Consider the edge of figure 4.1 and the accompanying function declaration of listing 4.4.

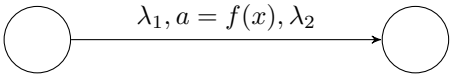


Figure 4.1: Example of how to handle function calls in assignments

```
1 void f(int i){
2     b = i*2;
3     return b;
4 }
```

Listing 4.4: Listing of the function f belonging to figure 4.1

If we parse the update statement of the edge according to the algorithm presented in definition 5 we get the following rewriting:

1. $\Lambda(\lambda_1, a = f(x), \lambda_2)$ (input)
2. $\Lambda(\lambda_1) +\!\!+ \Lambda(a = f(x)) +\!\!+ \Lambda(\lambda_2)$ (rule 5 applied 2 times)

   We leave out the first and last statement for clarity.

3. $\Lambda(f(x)) +\!\!+ \{a = \text{retVal}\}$ (rule 1c applied)
4. $\Lambda(i = x) +\!\!+ \Lambda((b = i + 2), (\text{return } b)) +\!\!+ \{a = \text{retVal}\}$ (rule 6 applied)
5. $\{i = x\} +\!\!+ \Lambda((b = i + 2), (\text{return } b)) +\!\!+ \{a = \text{retVal}\}$ (rule 1b applied)
6. $\{i = x\} +\!\!+ \Lambda(b = i + 2) +\!\!+ \Lambda(\text{return } b) +\!\!+ \{a = \text{retVal}\}$(rule 5 applied)
7. $\{i = x\} +\!\!+ \Lambda(v_1 = i, v_2 = 2) +\!\!+ \{b = v_1 + v_2\} +\!\!+ \Lambda(\text{return } b) +\!\!+ \{a = \text{retVal}\}$(rule 1e applied)
8. $\{i = x\} +\!\!+ \Lambda(v_1 = i)\Lambda(v_2 = 2) +\!\!+ \{b = v_1 + v_2\} +\!\!+ \Lambda(\text{return } b) +\!\!+ \{a = \text{retVal}\}$(rule 5 applied)
9. $\{i = x\} +\!\!+ \{v_1 = i\}\{v_2 = 2\} +\!\!+ \{b = v_1 + v_2\} +\!\!+ \Lambda(\text{return } b) +\!\!+ \{a = \text{retVal}\}$(rule 1a and 1b applied)
10. $\{i = x\} +\!\!+ \{v_1 = i\} +\!\!+ \{v_2 = 2\} +\!\!+ \{b = v_1 + v_2\} +\!\!+ \Lambda(\text{retVal} = b) +\!\!+ \{a = \text{retVal}\}$(rule 2 applied)

11. $\{i = x\} +\!\!+ \{v_1 = i\} +\!\!+ \{v_2 = 2\} +\!\!+ \{b = v_1 + v_2\} +\!\!+ \{\text{retVal} = b\} +\!\!+ \{a = \text{retVal}\}$(rule 1b applied)
12. $\{i = x\ , v_1 = i, v_2 = 2, b = v_1 + v_2, \text{retVal}\ = b, a = \text{retVal}\}$

If we replace $f(x)$ in figure 4.1 with $f(f(x))$ we show the possibility to have function calls as parameters (which is supported in UPPAAL). However this introduces no problems with our algorithms as after step 4, $\Lambda(i = x)$ then is replaced by $\Lambda(i = f(x))$ which can be parsed again (Note that UPPAAL itself does not support (recursive) function calls inside function calls, however our syntax (as mentioned in section 3.2.1) does allow this, which proves useful at this moment).

Conditional structures In order to eliminate the possibility to have function calls inside the conditional statement guarding the entrance of the conditional structure we have chosen to transfer the conditional check to an assignment and then later we only have to check the value of that variable. However, for while structures, this introduces another problem, because, if we transfer the conditional expression outside the while loop, for the second (and following evaluations) the variable used for the conditional check will not be assigned a new value again. Therefore, to solve this, we explicitly copy the assignment to the end of the body.
For example, if we look at listing 4.5 we see a simple while-loop that will print the numbers 0 to 9. If we move the conditional check outside the while-loop we get the listing of 4.6, which prints the number 0 infinitely many times. To solve this we, as explained, copy the assignment to v to the end of the body of the while-loop, resulting in listing 4.7, that has the same behaviour as the original listing.

```
1    i = 0;
2    while(i < 10){
3        print i;
4        i++;
5    }
```

Listing 4.5: Original code

```
1    i = 0;
2    v = i < 10;
3    while(v){
4        print i;
5        i++;
6    }
```

Listing 4.6: Rewriting, resulting in incorrect behaviour

```
1    i = 0;
2    v = i < 10;
3    while(v){
4        print i;
5        i++;
6        v = i < 10;
7    }
```

Listing 4.7: Correct rewriting

Initialisation of local parameters of a function (6) Consider the example function declaration as found in listing 4.8 which is called in figure 4.2 by $add(x, y)$. It could be (because of the return statement) that the variables i and j become relevant at the beginning of the function, however these variables are not used anywhere but inside the function. Actually the variables x and y that are passed to the function, as arguments in the function call, should become relevant. To cope with this we introduce a explicit initialisation assignment of these parameters. The effect of the function is not altered in this way but our algorithm now marks x and y relevant at the beginning of the function and the relevance can be passed on outside the function, as it should be.

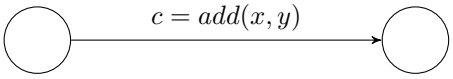


Figure 4.2: Example of how to handle function calls in assignments

```
1 int add(int i, int j){
2     return i+j;
3 }
```

Listing 4.8: Initialisation of local parameters - original code

The parsing of these statements looks as follows:

1. $\Lambda(c = \text{add}(x, y))$ (input)
2. $\Lambda(\text{add}(x, y)) +\!\!+ \Lambda(c = \text{retVal})$ (rule 1c applied)
3. $\Lambda(i = x) +\!\!+ \Lambda(j = y) +\!\!+ \Lambda(\text{return } i + j) +\!\!+ \Lambda(c = \text{retVal})$ (rule 6 applied)
4. $\Lambda(i = x) +\!\!+ \Lambda(j = y) +\!\!+ \Lambda(\text{retVal} = i + j) +\!\!+ \Lambda(c = \text{retVal})$ (rule 2 applied)
5. $\Lambda(i = x) +\!\!+ \Lambda(j = y) +\!\!+ \Lambda(v_1 = i, v_2 = j, \text{retVal} = v_1 + v_2) +\!\!+ \Lambda(c = \text{retVal})$ (rule 1e applied)
6. $\Lambda(i = x) +\!\!+ \Lambda(j = y) +\!\!+ \Lambda(v_1 = i) +\!\!+ \Lambda(v_2 = j)\Lambda(\text{retVal} = v_1 + v_2) +\!\!+ \Lambda(c = \text{retVal})$ (rule 5 applied two times)
7. $\{i = x\} +\!\!+ \{j = y\} +\!\!+ \{v_1 = i\} +\!\!+ \{v_2 = j\} +\!\!+ \{\text{retVal} = i + j\} +\!\!+ \{c = \text{retVal}\}$ (rule 1c applied 4 times)
8. $\{i = x, j = y, v_1 = i, v_2 = j, \text{retVal} = i + j, c = \text{retVal}\}$

4.1.2 Changed, used and directly used

In order to reason about variables in *guards* and *updates* we define a variable to be *changed* by a statement $\lambda \in \Lambda(e)$ if its value after λ can be different from its current value. A variable is *directly used* by an edge e if it is part of the guard of e or it can be *directly used* at a location if it is part of the invariant of a location. Also a variable is *used* if it is part of the update statement of e. In order to define the above we first introduce *vars*(*expr*) to denote which variables occur in an expression. *vars*(*expr*) is defined inductively as:

- vars(constant) = $\emptyset$

- vars(z) = z, where z is an integer, boolean, clock or array variable
- vars(expr$_1$ $\sim$ expr$_2$) = vars(expr$_1$) $\cup$ vars(expr$_2$)
- vars($-$expr) =vars(expr)
- vars($\neg$expr) =vars(expr)
- vars(channel[expr]) = vars(expr)

Based on the definition of variables in an expression we define which variables are *used* or *changed* by a statement λ, $used(\lambda)$ or $changed(\lambda)$, in table 4.2.

$\lambda \equiv if(\varphi)\{B_1\}$	$used(\lambda) = vars(\varphi) \cup used(B_1)$
$\lambda \equiv if(\varphi)\{B_1\}else\{B_2\}$	$used(\lambda) = vars(\varphi) \cup used(B_1) \cup used(B_2)$
$\lambda \equiv while(\varphi)\{B_1\}$	$used(\lambda) = vars(\varphi) \cup used(B_1)$
$\lambda \equiv y = expr$	$used(\lambda) = vars(expr)$
$\lambda \equiv B_1; B_2$	$used(\lambda) = used(B_1) \cup used(B_2)$

Table 4.1: The function *used*

$\lambda \equiv if(\varphi)\{B_1\}$	$changed(\lambda) = changed(B_1)$
$\lambda \equiv if(\varphi)\{B_1\}else\{B_2\}$	$changed(\lambda) = changed(B_1) \cup changed(B_2)$
$\lambda \equiv while(\varphi)\{B_1\}$	$changed(\lambda) = changed(B_1)$
$\lambda \equiv y = expr$	$changed(\lambda) = \{y\}$
$\lambda \equiv B_1; B_2$	$changed(\lambda) = changed(B_1) \cup changed(B_2)$

Table 4.2: The function *changed*

We define a special kind of *used*, *directly used*, to indicate that a variable is used in a guard, synchronization or an invariant.

- On an edge:
  - a is *directly used* in the guard of an edge e if $a \in used(guard(e))$.
  - a is *directly used* in the synchronisation part of an edge if $a \in vars(channel(e))$.
  - The set of all directly used variables of an edge e is $dir\_used(e)$
- At a location:
  - a is *directly used* at a location l if $a \in used(I(l))$.
  - The set of all directly used variables of l is $dir\_used(l)$

A problem with the linear process equations of [30] was that, by linearising the original processes into one linear form, the original control flow was lost. This control flow needed to be reconstructed first, therefore they included definitions of *source* and *destination* functions as well as a definition called *rules* to determine if a parameter is a *control flow parameter* or a *data parameter*. In UPPAAL the control flow can be directly derived from the process specifications. Note that [30] also mentions the existence of control flow information in the state parameters of the original specification. For the moment we assume the location variables are the only variables controlling the control flow of the program, they are similar to the Control Flow Parameters (CFPs) of [30],

whereas the other variables are the Data Variables (DPs), which can be divided into local and global variables.

Definition 6 (Local & Global variables). We have a network P of timed automata P_i. Let $a \in V_P \cup C_P$ be a variable, which is a *local variable* of P_i if all edges that change or use a are part of the set of edges E_{P_i}. A variable that is not a local variable of one of the timed automata in P is a global variable.

- a is local in P_i if $a \in \mathit{used}(P_i) \vee a \in \mathit{changed}(P_i) \wedge \forall j \cdot i \neq j \implies a \notin \mathit{used}(P_j) \wedge a \notin \mathit{changed}(P_j)$
- a is global in P if $\forall P_i \in P \cdot a \notin \mathit{local}(P_i)$

From this point we use the set A for referring to all variables, data or clock, global or local.

4.2 Relevance algorithm

The next step is the relevance algorithm itself. This algorithm will identify if a variable is relevant at a given location or not. If a variable is not relevant at a particular point it can be reset to its initial valuation. We first present a basic version of the algorithm and

Definition 7 (Relevance). For a network P of simple timed automata P_i, $1 \leq i \leq n$, with n the number of automata, variables $a, b \in A$ and a location $l \in L_{P_i}$. We use $(a, l) \in R$ (or $R(a, l)$) to denote that the value of a is relevant at location l. Formally R is the smallest relation such that:

1. If a is directly used in some $e \in E_P$, $a \in \mathit{dir\_used}(e)$ and $l = \mathit{src}(e)$ then $R_P(a, l)$
2. If a is directly used at some location l, $a \in \mathit{dir\_used}(l)$ then $R_P(a, l)$
3. If a is used in a property then: $\forall l \in L_P \cdot R_P(a, l)$
4. If $R_P(b, l')$, $\exists e \in E_P$ such that $\mathit{src}(e) = l$ and $\mathit{target}(e) = l'$, such that $a \in \mathit{processSeq}(\Lambda(\mathit{update}(e)), \{b\})$ then $R_P(a, l)$
5. If $R_P(b, l')$, $\exists e \in E_P$ such that $\mathit{src}(e) = l$, $l \in L_{P_i}, l' \in L_{P_j}$ and $i \neq j$, such that $a \in \mathit{processSeq}(\Lambda(\mathit{update}(e)), \{b\})$ then $R_P(a, l)$

Explanation of the algorithm For a better understanding we briefly describe each of the five clauses:

1. A variable is directly used on an edge if it is either used in a guard or in the synchronisation part of an edge. If that is the case this variable directly influences the control flow of the program and needs to be marked relevant at the source of the edge.
2. The same holds for a variable that is used in an invariant of a location. Because an invariant needs to be true at a location the values of the variables used in that invariant are also important making it necessary for the variables to be marked relevant.
3. A variable that is used in a property is automatically marked relevant at all locations of P. Because a change in the value of such a variable could cause a change in the truth value of the property therefore the values of these variables are relevant. (see section 4.3.3 for an improvement on this)
4. This clause takes care that if a variable is relevant at the target location of an edge that than every variable that is used in the rhs of an assignment to this relevant variable is also marked relevant. If there is no assignment then the relevant variable itself also becomes relevant at the source location as its value is already important at that point. Finally we mark variables relevant that are used in conditional statements (for instance the condition of an if-statement).
5. The last clause takes care of the same as the 4^{th} clause but not for variables that are relevant at the target location of the edge but instead for all global variables that are relevant at any point in a parallel automaton. Because of all the possible interleaving between automata the value of those relevant, global, variables are also relevant here.

Algorithm 1 processSeq(stmt_seq statements, variable relevant)

```
1: if (statements.isEmpty()) then
2:    return relevant
3: else
4:    relevant = processStat(statements.tail() , relevant)
5:    return processSeq(statements.withoutTail() , relevant)
6: end if
```

Algorithm 2 processStat(single_stmt stat , set⟨variables⟩ relevant)

```
1: if (stat ≡ u = expr) then
2:    if (u ∈ relevant) then
3:       removeVar(u, relevant)
4:       for all var ∈ used(expr) do
5:          insertVar(var, relevant)
6:       end for
7:    end if
8: else if (stat ≡ if(φ) B1 else B2) then
9:    for all var ∈ vars(φ) do
10:      insertVar(var, relevant)
11:   end for
12:   relevant = relevant ∪ processSeq(B1), relevant)
13:   relevant = relevant ∪ processSeq(B2), relevant)
14: else if (stat ≡ while(φ) B) then
15:   while (relevant ≠ temp) do
16:      temp = relevant
17:      for all var ∈ vars(condition) do
18:         insertVar(var, relevant)
19:      end for
20:      relevant = relevant ∪ processSeq(B, relevant)
21:   end while
22: end if
23: return relevant
```

Algorithm 3 insertVar(variable var, set⟨variables⟩ relevant)

```
1: relevant.insert(var)
```

The use of insertVar and removeVar At the moment the methods insertVar and removeVar (algorithms 3 and 4) simply insert the variables of the first parameter into the set mentioned as second parameter. However in section 4.3.1 we need to cope with inserting array variables into the relevant set. The insertion of array variables is more complex due to variable indices and possible function calls. Therefore we need a more advanced method at that point and as a precaution we already define (a basic version of) this method here to keep the original algorithm as much intact as possible.

Algorithm 4 removeVar(variable var, set⟨variables⟩ relevant)

1: relevant.remove(var)

Why do we need a While-loop in line 14? The necessity of the while-loop can be explained by the following UPPAAL code fragment:

```
1   ...
2   while (...) do
3       a=b;
4       b=c;
5   end while
6   ...
7   }
```

Listing 4.9: Example demonstrating the need of the while-loop

We now assume a is relevant at line 6 of this piece of code. If we now run the algorithm we see (without the while-loop of line 14 of the algorithm) that b is added by the algorithm to the set of relevant variables. However it is possible that the body of the while loop of line 2-5 is executed twice and then c can also become relevant. Therefore we have to determine the (smallest) fix-point of the set of relevant variables for this UPPAAL while-loop and therefore the while-loop of line 14 is required.

The above definitions allow us to declare a variable relevant (at a particular location) according to a specification of (a network of) timed automata. In section 5 we define a transformation that transforms a network of timed automata into another network of timed automata. We will show that these two networks of timed automata are bisimilar. To prove this bisimilarity we need to take a look at the corresponding state spaces of these networks. A state space is defined as a timed labelled transition system (TLTS) and therefore we take the definition of relevance at a location in a timed automata to a definition of relevance in a state of a TLTS. But first we recall some basic definition on a *state* and introduce an auxiliary function to easily cope with the valuation of both clock and data variables.

Definition 8 (State). Given a network P of simple timed automata and a corresponding instantiation of it in the form of a timed labelled transition system $T = \langle S, s_0, \rightarrow \rangle$ a state $s = \langle \bar{l}, \sigma, \omega \rangle \in S$ of this TLTS is defined as:

- $\bar{l}$ is the vector $(l_1, \ldots, l_n) \in (L_1 \times \ldots \times L_n)$ of locations with each location location l_i being the current location of process P_i with $1 <= i <= n$.
- $\sigma : C \rightarrow \mathbb{R}_{\geq 0}$ is a clock valuation from the set of clock valuations $\mathscr{C}$.
- $\omega : V \rightarrow \mathbb{Z}$ is a variable valuation from the set of variable valuations $\mathscr{V}$.
- Because our algorithm does not distinguish between a clock variable and a discrete variable we have introduced the set A, which is a combination of C and V. The valuation of this set A is given by:

$$\text{val}(a) = \begin{cases} \sigma(a) & \text{if } a \in C, \\ \omega(a) & \text{if } a \in V. \end{cases}$$

Based on the previous definition of relevance for timed automata we introduce a definition of relevance for timed labelled transition systems.

Definition 9 (**Relevance in states**)**.** Given a network P of simple timed automata with a relevance relation R_P and a corresponding timed labelled transition system $T_P = \langle S_P, s_P^0, \rightarrow_P \rangle$. The relevance of a variable $a \in A$, given a state $s = \langle \bar{l}, \sigma, \omega \rangle \in S_P$, denoted $\mathit{Relevant}(a, s)$, is defined by:

$$\mathit{Relevant}(a, s) = \bigvee_{i \in \{1..n\}} R_P(a, l_i)$$

We claim that the value of a variable that is not relevant, in a state, does not matter. Therefore, given a TLTS $T = \langle S, s^0, \rightarrow \rangle$ we introduce a relation $\cong$ on states, given by:

$$s \cong s' \Leftrightarrow \bar{l}_s = \bar{l}_{s'} \wedge \forall a \in A : (\text{Relevant}(a, s) \Rightarrow \text{val}_s(a) = \text{val}_{s'}(a)), \text{ where } s, s' \in S$$

We will prove that it is a strong bisimulation by introducing a couple of lemma's which we use to prove the theorem that the relation $\cong$ is a strong bisimulation. We adopt the definition of strong timed bisimulation as also defined in [24].

Strong timed bisimulation Two states s_0 and s_0' are strong timed bisimilar ($s \sim s'$) if the following holds:

- $\forall s_1 :$ if $s_0 \xrightarrow{a} s_1$ then $\exists s_1' : s_0' \xrightarrow{a} s_1'$ and $s_1 \sim s_1'$
- $\forall s_2 :$ if $s_1 \xrightarrow{a} s_0$ then $\exists s_0' : s_1' \xrightarrow{a} s_0'$ and $s_1 \sim s_1'$
- $\forall s_1 :$ if $s_0 \xrightarrow{\delta} s_1$ then $\exists s_1' : s_0' \xrightarrow{\delta} s_1'$ and $s_1 \sim s_1'$
- $\forall s_2 :$ if $s_1 \xrightarrow{\delta} s_0$ then $\exists s_0' : s_1' \xrightarrow{\delta} s_0'$ and $s_1 \sim s_1'$

Before we can define these lemma's we first need some declarations that we are going to use in these lemma's and we present these declarations first:

- Below we assume we have a network P of simple timed automata, variables $a, b \in A$, locations $l, l' \in L_P$ and a relevance relation R_P.
- From this network P a timed labelled transition system (TLTS) T can be derived. This T is denoted by $\langle S, s^0, \rightarrow \rangle$.
- States $s, s' \in S$ are defined as $s = \langle \bar{l}, \sigma, \omega \rangle$ and $s' = \langle \bar{l}', \sigma', \omega' \rangle$ with valuation functions $val(a)$ and $val'(a)$ respectively. s_0, s_0', s_1, s_1' and s'' are defined analogue.

Lemma 10. Let s and s' be two states such that $s \cong s'$, and $\mathit{Relevant}(a, s')$ for some a. Then it follows that $\mathit{Relevant}(a, s)$.

Proof. Assume that $s \cong s'$ and $\mathit{Relevant}(a, s')$. By definition of *Relevant*, we have $R(a, l_i')$. By the definition of $\cong$ we have $l_i = l_i'$. Since $R(a, l_i')$ this immediately implies $R(a, l_i)$ and we get $R(a, s)$

□

Lemma 11. The relation $\cong$ is an equivalence relation.

Proof. Reflexivity is trivial. For symmetry, we assume $s \cong s'$. For all a, if $\mathit{Relevant}(a, s')$, then by Lemma 10 also $\mathit{Relevant}(a, s)$. Therefore, by definition of $\cong$ and the assumption that $s \cong s'$, we obtain $\mathit{val}(a) = \mathit{val}'(a)$. Secondly, because of $s \cong s'$ we have $\bar{l} = \bar{l}'$. Combining these two we get $s' \cong s$.

For transitivity, assume $s \cong s'$ and $s' \cong s''$. If $\mathit{Relevant}(a, s)$, then by definition $\mathit{val}(a) = \mathit{val}'(a)$. Using symmetry and Lemma 10 it follows that $\mathit{Relevant}(a, s')$, and hence $\mathit{val}'(a) = \mathit{val}''(a)$ and $\mathit{val}(a) = \mathit{val}''(a)$. Also we have $\bar{l} = \bar{l}'$ and $\bar{l}' = \bar{l}''$ and thus $\bar{l} = \bar{l}''$. Therefore, $s \cong s''$ □

The next step is to show that if an edge e is enabled given some state s, then it is also enabled given a state s' such that $s \cong s'$.

Lemma 12. Let s and s' be states such that $s \cong s'$. Let $e \in E_P$, be an edge. Then $\mathit{guard}(e)(s) \iff \mathit{guard}(e)(s')$.

Proof. We need to show that for all $z \in \mathit{vars}(\mathit{guard}(e))$ it holds that $\mathrm{val}(a) = \mathrm{val}'(a)$. Assume that $\mathit{guard}(e)(s)$ holds. Let an arbitrary $a \in \mathit{vars}(\mathit{guard}(e))$ be given and $l = \mathit{src}(e)$. Since a is directly used in e it follows from Definition 7 that $R(a, l)$. This means, by Definition 9 that $\mathit{Relevance}(a, s)$. Using the definition of $\cong$ we obtain $\mathit{val}(a) = \mathit{val}'(a)$. Since a was chosen arbitrary, this holds for all $a \in \mathit{vars}(\mathit{guard}(e))$, so $\mathit{guard}(e)(s')$ also holds. □

Lemma 13. Let s and s' be states such that $s \cong s'$. Let $e \in E_P$, be an edge. Then $\mathit{channel}(e)(s) = a$ implies $\mathit{channel}(e)(s') = a$.

Proof. This proof is identical to the proof of Lemma 12 only substituting guard(e) by channel(e). □

Now that we have shown that enabledness of edges is conserved in the equivalence relation and that the ability to synchronise is conserved, the next step is to show that the equivalence relation also holds for successors of a state, therefore we have the following lemma. However before we present the next lemma we first introduce a notation to represent taking an edge.

Representation of taking an edge In order to represent that from a given state s we take an edge e we use the following notation: $s.\mathrm{take}(e)$. The semantics of *take* corresponds with the semantics of the action transition of section 3.2.3. Because both guards and invariants are side-effect free $s.\mathrm{take}(e)$ can be replaced by $[\![\mathrm{update}(e)]\!](s)$

Lemma 14. Let s and s' be states such that $s \cong s'$. Let $e \in E_P$, be an edge. Then $\mathit{guard}(e)(s)$ implies $s.\mathit{take}(e) \cong s'.\mathit{take}(e)$.

Proof. By definition of $\cong$, it has to be shown that for all variables a such that $\mathit{Relevant}(a, s.\mathit{take}(e))$ it holds that $\mathit{val}_{s.\mathit{take}(e)}(a) = \mathit{val}_{s'.\mathit{take}(e)}(a)$.

We prove by induction on single_stat that this is indeed the case. We distinguish the four different cases of single_stat: an assignment, an if-else statement, a while statement or a sequential statement. For each kind of statement we have to proof that for all variables a such that $\mathit{Relevant}(a, s.\mathit{take}(\mathit{single\_stat}))$

it holds that $val_{s.take(single\_stat)}(a) = val_{s'.take(single\_stat)}(a)$.

- Single_stat $\equiv c =$ expr
  - If $c \equiv a$ the value of a changes and to have $val_{s.take(e)}(a) = val_{s'.take(e)}(a)$ we have to prove that $\forall b \in vars(expr) \cdot val(b) = val'(b)$. Because of line 4 of algorithm 2 we get $Relevant(b, s)$. Combined with $s \cong s'$ and the definition of $\cong$ we get $val_s(b) = val_{s'}(b)$.
  - If $c \neq a$ the value of a does not change and because of $s \cong s'$ we have $val(a) = val'(a)$ and we get $val_{s.take(e)}(a) = val_{s'.take(e)}(a)$
- Single_stat $\equiv$ if(φ)B_1 else B_2. We use the induction hypothesis that the lemma holds for a single_stmt to proof that the lemma holds for both B_1 and B_2. We only have to proof that the result of the condition check is the same for s and s'. If we have an arbitrary $b \in$ vars(φ) we have to show that $\text{val}_s(b) = \text{val}_{s'}(b)$ in order for $[\![\varphi]\!](s) = [\![\varphi]\!](s')$ to hold. Because of algorithm 2 lines 17/18 we have $R(b, s)$ and combined with $s \cong s'$ and the definition of $\cong$ we have $\text{val}_s(b) = \text{val}_{s'}(b)$.
- Single_stat $\equiv$ while(φ)B_1. Since the body (B_1) of this while-statement can be executed 0 to infinite times we use another induction this time on the number of iterations of the while-loop of the algorithm. We show that it holds for 0 iterations, consequently assume it holds for n iterations and we finally show that it also holds for $n + 1$ iterations:
  - If the body of the while-statement is executed 0 times we know that the only part that is executed is the check of the condition and no other statements are executed. Because this condition check can have no side effects we know that if we have a state $s = \langle \bar{l}, \sigma, \omega \rangle$ that for the state s.take(single_stat) $= \langle \bar{l'}, \sigma', \omega' \rangle$ it holds that $\sigma' = \sigma$ and $\omega' = \omega$.
    Secondly because in lines 12-17 of algorithm 2 we know that the relevant set at the end of this while-statement is a subset of the relevant set at the beginning of this while-statement.
    Concluding we know that for every variable a such that Relevant(a, s.take(single_stat)) that also Relevant(a, s) holds and we know that $\text{val}_s(a) = \text{val}_{s.\text{take(single\_stat)}}(a)$ completing this part of the proof.
  - Using the second induction hypothesis we know that the lemma holds for n iterations of the while-statement. We show that it holds for $n+1$ statements. Using the set union of line 16 of algorithm and the first induction hypothesis, that the lemma holds for a single_stat (the statements of the body can be written as a single_stat), we know that if $s.\text{take}^n$(body (this is the second induction hypothesis) holds that also $s.\text{take}^n$(body.take(body)) (which is $s.\text{take}^{n+1}$(body)) holds.
  - The last part that is missing is whether we know that the relevant variables are in the R's in the proof above. However we know that every relevant variable should be in R because of the condition in line 15 of algorithm 2. We know that it is not possible that a relevant variable is not in R otherwise the algorithm would not have terminated.

- Single_stat $\equiv B_1; B_2$. Using the induction hypothesis for B_1 we know that, given $s \cong s'$, that $s.\text{take}(B_1) \cong s'.\text{take}(B_1)$ holds. Using the induction hypothesis again for B_2 we also know that $s.\text{take}(B_1).\text{take}(B_2) \cong s'.\text{take}(B_1).\text{take}(B_2)$ holds. Finally we substitute $\text{take}(B_1).\text{take}(B_2)$ by $\text{take}(\text{single\_stat})(a)$ and the proof is complete.

□

The last thing to show is that the equivalence relation preserves the validity of the invariant constraints of the target location. So if a target location does not violate an invariant its equivalent also does not violate an invariant. This is expressed in the following lemma.

Lemma 15. Let s and s' be states such that $s \cong s'$, then $s \models I \iff s' \models I$.

Proof. Because of $s \cong s'$ we have $\bar{l} = \bar{l'}$, therefore we have to show that $\forall l \in \bar{l}$ it holds that $s \models I(l) \iff s' \models I(l)$.
To show this we have to proof $\forall a \in \text{vars}(I(l))$ that $\text{val}(a) = \text{val}'(a)$. By definition 7 we know $R(a, l)$ and thus by definition 9 we have $\text{Relevant}(a, s)$. Therefore we get, because of $s \cong s'$, that $\text{val}(a) = \text{val}'(a)$ □

Using the lemmas above we can easily prove the following theorem.

Theorem 16. The relation $\cong$ is a strong timed bisimulation.

Proof. Let s_0 and s'_0 be states such that $s_0 \cong s'_0$. Also assume that $s_0 \to s_1$. Because $\cong$ is symmetric (Lemma 11), we only need to prove that there exists a transition $s'_0 \to s'_1$ such that $s_1 \cong s'_1$. By the operational semantics there is an edge e such that $\mathit{guard}(e)(s_0)$ holds and $s_1 = s_0.\mathit{take}(e)$. By Lemma 12 we know that $\mathit{guard}(e)(s'_0)$ holds. Using Lemma 13 we also know that $\mathit{channel}(e)(s_0) = \mathit{channel}(e)(s'_0)$ so synchronisation, if necessary, takes place with the same other process. Therefore, $s'_0 \to s'_0.\mathit{take}(e)$. By Lemma 14 $s_0.\mathit{take}(e) \cong s'_0.\mathit{take}(e)$ and we know that $s'_0.\mathit{take}(e)$ is a valid state because $s_0.\mathit{take}(e) \models I$ and by Lemma 15 we know that $s_0.\mathit{take}(e) \models I \iff s'_0.\mathit{take}(e) \models I$.

The second part of the strong timed bisimulation requires that if $s_0 \xrightarrow{\delta} s_1$ that also $s'_0 \xrightarrow{\delta} s'_1$ and that $s_1 \cong s'_1$ holds. The only thing that can hold back a delay step is an invariant but if a clock variable c is used in the invariant of one of the locations of $\bar{l_0}$ ($\bar{l_0} = \bar{l'_0}$ we know by clause 2 of algorithm 7 that $\text{Relevant}(c, s_0)$ and $\text{Relevant}(c, s'_0)$ both hold ensuring that $\text{val}_{s_0}(c) = \text{val}_{s'_0}(c)$. So we know that if s_0 can do the delay step, s'_0 can also do the delay step. We also know that the semantics of urgent/committed are not violated because the set of enabled edges of both s and s' are the same. This is because for every edge e we have $\text{guard}(e)(s) = \text{guard}(e)(s')$. Finally the values of (at least) the relevant variables are the same because both have the same value before the delay step and get both increased by the same delay $\delta \in \mathbb{R}_{\geq 0}$ resulting in $s_1 \cong s'_1$. □

4.3 Improvements and extensions

4.3.1 Arrays

For normal variables the theory presented before is sufficient and no problems arise, but if we look at extending the theory, to include array variables, a couple

of problems arises. For arrays with constants as indices things are relatively normal and the same theory as for normal variables applies. However for arrays with non-constant indices everything gets more complicated. The second problem arises when one of the index-expressions contains a function call, however this is not possible for simple timed automata as we remove any function calls from array indices in the function Λ, see definition 5.
Therefore we update the functions insertVar and removeVar, which serve as a replacement for respectively insertVar and removeVar from algorithm 2. Both methods return a set of relevant variables.

Algorithm 5 insertVar(variable var, set⟨variable⟩ relevant)

```
1:  if var.type == ARRAY then
2:    for (index_1 to index_n of var) do
3:      if (index_i != CONSTANT) then
4:        for all var ∈ vars(index_i) do
5:          relevant = insertVar(var, relevant)
6:        end for
7:        for (∀j ∈ range_i) do
8:          tempVar = var, with the i^th index replaced by j
9:          relevant = insertVar(tempVar, relevant)
10:       end for
11:     end if
12:     return relevant
13:   end for
14: end if
15: relevant.insert(var)
16: return relevant
```

Algorithm 6 removeVar(variable var, set⟨variable⟩ relevant)

```
1:  bool isConstant = true
2:  if var.type == ARRAY then
3:    for (index_1 to index_n of var) do
4:      if (index_i != CONSTANT) then
5:        isConstant = false
6:        for all var ∈ vars(index_i) do
7:          insertVar(var, relevant)
8:        end for
9:      end if
10:   end for
11: end if
12: if isConstant then
13:   relevant.remove(var)
14: end if
15: return relevant
```

The correct handling of array variables One thing to notice is that the used method is a great over-approximation, however we do not have any other

option without looking at the values of variables. Using these two methods we can ensure that no variable that should be relevant is not marked relevant. First, for the insertVar method, we check each index (from left to right) if it contains a constant value or a variable. In the case of a constant 'nothing' happens and the variable is inserted. In the case of a variable we 'simply' call the insertVar method again for all possible values of this variable. Because we do this for all values we can guarantee that the correct one(s) is/are included.
Secondly, for the removeVar method, we are dealing with the opposite case. We do not want to remove any relevant variable of which we are not sure it is not relevant. Therefore, if one or more of the indices contain a variable, we do not remove any variable from the set of relevant variables, as we can not tell which one we need to delete. Using both methods we can be certain that every relevant variable is marked relevant.

Example of inserting an array variable As an example we show how the algorithm behaves if we have determined that $a[b][1]$ is relevant. Assume the array is declared as int $a[3][3]$ and that relevant is empty at the beginning of the example.
In the first iteration of the first for-loop (line 2) the condition of line 3 evaluates to true and in line 4/5/6 the method is recursively called for the variable b resulting in b being added to the set relevant. In lines 7-10 consequently the following (recursive) method calls are executed:

- insertVar(a[0][1], relevant)
- insertVar(a[1][1], relevant)
- insertVar(a[2][1], relevant)

For each of these methods the for-loop of line 2 consists of two iterations. In each iteration the condition of line 3 evaluates to false and when line 15 is reached the variable used in the method call is inserted into the set relevant. At the end the set relevant consists of the following variables: {b,a[0][1],a[1][1],a[2][1]}

4.3.2 Value passing variables

One of the greatest issues when marking variables relevant is that in UPPAAL global variables immediately become relevant at (almost) every location. The cause of this is the problem that you can not predict the interleaving of the processes. Therefore a global variable that seems to be not relevant in one process can become relevant because it is relevant at another process. There is however one specific category of variables that is worth taking another look at and this kind of global variables we call *value passing variables*. Because UPPAAL does not support direct value passing during synchronisation, value passing has to be achieved through the use of global variables. One of the UPPAAL tutorials [5] also mentions these value passing variables as a modelling pattern. Instead of value passing variables they mention these variables as shared variables and they describe it as:

'The general idea is that a sender and a receiver synchronise over shared binary channels and exchange data via shared variables. Since UPPAAL evaluates the assignment of the sending synchronisation first, the sender can assign a value to the shared variable which the receiver can then access directly'

If a global variable is only used as a value passing variable to pass on values between two processes and nowhere else then its value is only relevant during the execution of the update statements of both edges. As a consequence the value of the value passing variable is not relevant outside the synchronisation and therefore we should reset its value in order to achieve a minimal state space. We first define which global variables we consider to be part of the set of value passing variables.

Definition 17 (**Value passing variable**)**.** The set of value passing variables is a subset of the set of global variables. A global variable a is a value passing variable if it is used only in update statements and:

$$\forall e \in E \text{ s.t. } a \in \text{used}(\text{update}(e)) \cdot$$

$$\exists z \in \Sigma \cdot (z? = \text{ channel}(e) \wedge \forall e' \in E \cdot z! = \text{ channel}(e') \Rightarrow v \in \text{ changed}(e'))$$

in combination with:

$$\forall e \in E \text{ s.t. } v \in \text{changed}(\text{update}(e)) \cdot$$

$$\exists z \in \Sigma \cdot (z! = \text{ channel}(e) \wedge \forall e' \in E \cdot z? = \text{ channel}(e') \Rightarrow v \in \text{ used}(e'))$$

The following problem that arises is how do we incorporate the above definition into our relevance algorithm in such a way that value passing variables are not unnecessary marked relevant, that they (eventually) get reset by the transformation algorithm of chapter 5 and finally that the relevance is correctly passed from the sending side to the receiving side of the value passing variable.

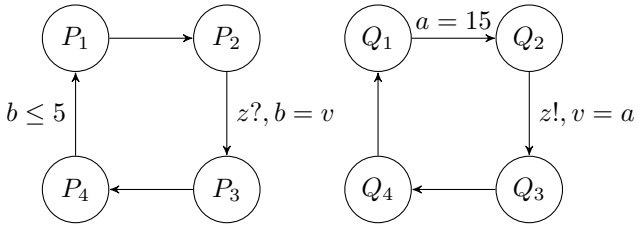


Figure 4.3: Example used to demonstrate the idea of value passing variables

Figure 4.3 shows a example of two processes that use a value passing variable s to transfer the value of local variable a (of process Q) to local variable b (of process P). For variable b there are no problems, because it gets marked relevant at P_4 because of the guard on the incoming edge, consequently also at P_3, but not at P_2, because of the assignment on the transition $P_2 \rightarrow P_3$.
For variable v it is a bit more complicated. Using the original algorithm, without the notion of value passing variables, v would be marked relevant at state P_2 because of the use in the right hand side of an assignment to a relevant variable.

Consequently, due to its global nature, v would also be marked relevant at states P_1, P_4, P_3 and the states Q_1, Q_3 and Q_4. However, because v is a value passing variable it is always the case that the value of v is changed before that it is used, so the value of v is never relevant at any location but only right in between the execution of the sending and receiving edge. Using the current approach of marking variables relevant at locations does not suffice any more and therefore we add a set containing pairs of variables and channels. By marking a variable relevant at a channel we avoid marking value passing variables relevant at locations even though they are never relevant at those locations.
Finally, because of the relevance of v in channel z, a will be marked relevant at $Q2$ because it is used in the right hand side of an assignment to a relevant variable.

Updated algorithm The original definition (def. 7) needs a couple of adaptations, but algorithms 1 and 2 do not need any adaptations at all. Basically we only have to divide the relevance set into a relevance (at location) set and a relevant-at-channel set resulting in the following definition.

Definition 18 (**Relevance including value passing variables**)**.** For a network P of simple timed automata P_i, $1 \leq i \leq n$, with n the number of automata, variables $a, b \in A$, a location $l \in L_{P_i}$ and a channel $z \in \Sigma_P$. We use $(a, l) \in R$ (or $R(a, l)$) to denote that the value of a is relevant at location l and we use $(a, z) \in \text{RelVP}$ (or $\text{RelVP}(a, z)$) to denote that the value of a is relevant at the receiving side of channel z. Formally R and RelVP are the smallest relations such that:

1. If a is directly used in some $e \in E_P$, $a \in \mathit{dir\_used}(e)$ and $l = \mathit{src}(e)$ then $R_P(a, l)$
2. If a is directly used at some location l, $a \in \mathit{dir\_used}(l)$ then $R_P(a, l)$
3. If a is used in a property then: $\forall l \in L_P \cdot R_P(a, l)$
4. If $R_P(b, l')$, $\exists e \in E_P$ such that $\mathit{src}(e) = l$ and $\mathit{target}(e) = l'$, such that $a \in \mathit{processSeq}(\Lambda(\mathit{update}(e)), \{b\})$ then:
   - $\text{RelVP}_P(a, \text{channel}(e))$, if a is a value passing variable
   - $R_P(a, l)$, otherwise
5. If $R_P(b, l')$, $\exists e \in E_P$ such that $\mathit{src}(e) = l$, $l \in L_{P_i}, l' \in L_{P_j}$ and $i \neq j$, such that $a \in \mathit{processSeq}(\Lambda(\mathit{update}(e)), \{b\})$ then:
   - $\text{RelVP}_P(a, \text{channel}(e))$, if a is a value passing variable
   - $R_P(a, l)$, otherwise
6. If $\text{RelVP}_P(b, z)$, $\exists e \in E_P$ such that $\mathit{src}(e) = l$ and $\mathit{channel}(e) = z!$, $l \in L_{P_i}$, such that $a \in \mathit{processSeq}(\Lambda(\mathit{update}(e)), \{b\})$ then:
   - $\text{RelVP}_P(a, \text{channel}(e))$, if a is a value passing variable
   - $R_P(a, l)$, otherwise

Correctness of the updated algorithm. Basically there are only some slight differences between this algorithm and the original algorithm of definition 7 and all these differences rely on the assumption that the value of value passing variables is only relevant during synchronisation and not in any other location.

The definition of value passing variables (definition 17) ensures this because a value passing variable is only used on receiving edges whereas every value passing variable is always changed on the corresponding sending edge.
The main difference is that a value passing variable is not marked relevant, in clauses 4 and 5 of the definition, at the source of the edge, causing that the variable becomes relevant at (almost) every other location due to its global nature. But instead the new definition only marks the variable relevant at the channel z. Secondly the variables that are marked relevant should be considered as relevant variables on the receiving edge of the channel z and that is what clause 6 of the definition ensures.
Now that we have shown how we cope with synchronisations in UPPAAL we will proof theorem 16 for the third type of transition, as described in section 3.2.3, the sync transition.

Theorem 19. The relation $\cong$ is a strong timed bisimulation

Proof. For the action transition and the delay transition this is already proven by theorem 16. Now we prove it for the synchronisation transition.
Let s_0, s_0', s_1, s_1' be states such that $s_0 \cong s_0'$. Assume that from s_0 we can take edges $e_0 : l_i \xrightarrow{v!} l_i'$ and $e_1 : l_j \xrightarrow{v?} l_j'$ resulting in a synchronising transition to s_0'. Because $\cong$ is symmetric we only need to prove that from s_1' we can take the same edges e_0 and e_1 resulting in a transition to s_1' such that $'s_0 \cong s_1'$. By the operation semantics guard$(e_0)(s_0)$ and guard$(e_1)(s_0)$ both hold and $s_1 = s_0$.take(e_0).take(e_1). By Lemma 12 we know that both guard$(e_0)(s_0')$ and guard$(e_1)(s_0')$ hold. Using Lemma 13 we also know that channel$(e_0)(s_0) =$ channel$(e_0)(s_0')$ and channel$(e_1)(s_0) =$ channel$(e_1)(s_0')$, so the synchronising transition can also take place in s_0'. By repeatedly applying Lemma 14 we first get $s_0.take(e_0) \cong s_0'.take(e_0)$ and consequently $s_0.take(e_0).take(e_1) \cong s_0'.take(e_0).take(e_1)$. Finally Lemma 15 gives us that $s_0.take(e_0).take(e_1) \models I \iff s_0'.take(e_0).take(e_1) \models I$.

□

4.3.3 Variable relevance in properties

One of the reasons for a variable to become relevant at some locations is that the variable is being used in a property (see clause 3 in definition 7). There are two possible solutions which both deal with the problem which variables to mark relevant because of a property.

1. Every referenced variable in a property is automatically relevant in all locations. For a local variable at all locations of its process. For a global variable at all locations of all processes.
2. Try to mark (some) variables only relevant for certain locations.

The first solution is the one that we presented originally in definition 7 and is very basic and most of the time an over-approximation because it marks every variable, that is used in the property, relevant at every location. The extension we are going to look at aims at improving the approximation by reducing the set of locations for which a variable is marked relevant.
The table below gives some examples about properties and their ideal result

Property	Ideal result
$x > 4$	x always relevant
$P_1.x > 4$	x always relevant in P_1
$P.l_1 \Rightarrow P.x > 4$	x relevant at $P.l_1$
$\neg P.l_1 \vee P.x > 4$	x relevant at $P.l_1$
$P.l_1 \wedge P.x > 4$	x relevant at $P.l_1$

Table 4.3: Examples of properties

In the table above the first and second row are clear as the value of x is always relevant in every state. However why can we say that x is only relevant at $P.l_1$ in the other 3 possibilities?
The idea we are going to use is based on the truth values of the property (sub)expression(s). If you take a look at table 4.4 you will see the truth table for several logical formulae.

N	M	$N \vee M$	$N \wedge M$	$N \Rightarrow M$
true	true	true	true	true
true	false	true	false	false
false	true	true	false	true
false	false	false	false	true

Table 4.4: Truth table for OR, AND and IMPLIES

The advantage we have in UPPAAL is that properties can contain location variables, variables telling at which location a specific process is. For instance the property $P.l_1 \Rightarrow P.x > 4$ ensures that if process P is in location l_1 the (local) variable (of P) x needs to be greater then 4. This also means that if process P is in any other location then l_1 the value of x is not (directly) important. In order to better reason about this subject we first give another truth table. This time with the same logical formulae but for each formulae we leave the value of one of the operands unfixed and then look if the resulting value can be given.

N	M	$N \vee M$	$N \wedge M$	$N \Rightarrow M$
true	*	true	*	*
false	*	*	false	true
*	true	true	*	true
*	false	*	false	*

Table 4.5: Truth table for OR, AND and IMPLIES with one truth value known

Looking at this, adapted, truth table we can see that for each logical formula we are (sometimes) able to determine the resulting value regardless of knowing the values of all the operands. In order to reason more easily we assume the logical formula of the property is transformed into a parse tree, which has as nodes the operands of the sub-properties and as leafs either an expression containing

variables (data or clock) or a location variable.
Of these two types of leafs the location variables are the only one of which we can say something, namely: If process P is at location l then the location variable l is true, while all other location variables of process P are false. As in the original algorithm we would mark all variables, that occur in a property, relevant for all locations. Using the knowledge presented here we can 'predict' for (some) locations the truth value of a (sub)property, resulting in a variable being not relevant for the truth value of a property for some locations.
Below we present our algorithm (algorithm 7) that, using a depth first search on the (binary) parse tree of the property, determines for each (sub)property if we can say (using the current knowledge) if this sub-property is always true or always false. For an OR-sub-property we can say it is always false if all its children (including sub-properties) are false and that it is always true if only one of his children is true. For an AND-sub-property it is exactly the opposite as this sub-property is always true if all its children are true and it is always false if only one of its children is false.

Algorithm 7 markNodes(Property prop)

if (prop $\equiv N \vee M$) **then**
  prop.trueLocs = N.trueLocs $\cup$ M.trueLocs
  prop.falseLocs = N.falseLocs $\cap$ M.falseLocs
else if (prop $\equiv N \wedge M$) **then**
  prop.trueLocs = N.trueLocs $\cap$ M.trueLocs
  prop.falseLocs = N.falseLocs $\cup$ M.falseLocs
else if (prop $\equiv \neg N$) **then**
  prop.trueLocs = N.falseLocs
  prop.falseLocs = N.trueLocs
else if (prop $\equiv l$) **then**
  prop.trueLocs = {l}
  prop.falseLocs = $L_{P_i} - \{l\}$, with L_{P_i} such that $l \in L_{P_i}$
else
  prop.trueLocs = {}
  prop.falseLocs = {}
end if

Now that we know, using algorithm 7, which sub-properties are always false or always true for certain locations we run another depth first search (algorithm 8) that looks at which variables are used in a sub-property and marks them relevant for all locations except the locations for which we know that the sub-property is always false or always true. Both algorithms are called from algorithm 9 which gives us a set R containing pairs of a location and a variable. The third clause of definition which serves as input for definition 7, instead of the third clause of that definition. The third clause now becomes:

3. If there exists a prop $\in$ properties such that $a \in$ propertyRelevance(prop) then $R(a, l)$

Algorithm 8 getRelevantFromProperty(Property prop)

```
if (prop ≡ N ∨ M or prop ≡ N ∧ M ) then
  N.trueLocs = N.trueLocs ∪ prop.trueLocs
  M.trueLocs = M.trueLocs ∪ prop.trueLocs
  N.falseLocs = N.falseLocs ∪ prop.falseLocs
  M.falseLocs = M.falseLocs ∪ prop.falseLocs
  R = getRelevantFromProperty(N) ∪ getRelevantFromProperty(M)
else if (prop ≡ ¬N) then
  N.trueLocs = N.trueLocs = ∪ prop.trueLocs
  N.falseLocs = N.falseLocs = ∪ prop.falseLocs
  R =getRelevantFromProperty(N)
else if (prop ≡ l) then
  Nothing has to be done
else
  for (each l ∈ L) do
    if (l ∉ prop.trueLocs ∪ prop.falseLocs) then
      for (each a ∈vars(prop)) do
        Add (a, l) to R
      end for
    end if
  end for
  return R
end if
```

Algorithm 9 propertyRelevance(Property prop)

```
markNodes(prop)
return getRelevantFromProperty(prop)
```

Correctness of algorithm including property relevance Instead of marking all the used variables at all location we now mark the used variables relevant at a subset of the locations. In order to reason about if this extension of the algorithm is correct we first need to define what we consider to be correct. What we want is that the outcome of the property check is always the same and that every variable that is not marked relevant by the above algorithm can be assigned any value without influencing the truth value of the property.

$$\forall \text{prop } \in \text{properties } \cdot \text{prop}(s) == \text{prop}(s')$$

In the above equation s' is equal to s, but for each variable of s that is not relevant we can assign any (random) value and we should not alter the outcome of the property.
We can say this is true because every variable is marked relevant in algorithm 8 for every location unless we are certain that the sub-property it belongs to is always true or always false in a specific location due to the known truth value of the location variables occurring in that sub-property. So by construction of algorithms 7, 8 and 9 we know it is correct.

4.4 Complete relevance algorithm

After the various adoptions to the original relevance algorithm we give an overview of the resulting algorithm:

Definition 20 (**Final relevance definition**)**.** For a network P of simple timed automata P_i, $1 \leq i \leq n$, with n the number of automata, variables $a, b \in A$, a location $l \in L_{P_i}$ and a channel $z \in \Sigma_P$. We use $(a, l) \in R$ (or $R(a, l)$) to denote that the value of a is relevant at location l and we use $(a, z) \in \text{RelVP}$ (or $\text{RelVP}(a, z)$) to denote that the value of a is relevant at the receiving side of channel z. Formally R and RelVP are the smallest relations such that:

1. If a is directly used in some $e \in E_P$, $a \in \mathit{dir\_used}(e)$ and $l = \mathit{src}(e)$ then $R_P(a, l)$
2. If a is directly used at some location l, $a \in \mathit{dir\_used}(l)$ then $R_P(a, l)$
3. If there exists a prop $\in$ properties such that $a \in$ propertyRelevance(prop) then $R(a, l)$
4. If $R_P(b, l')$, $\exists e \in E_P$ such that $\mathit{src}(e) = l$ and $\mathit{target}(e) = l'$, such that $a \in \mathit{processSeq}(\Lambda(\mathit{update}(e)), \{b\})$ then:
   - $\text{RelVP}_P(a, \text{channel}(e))$, if a is a value passing variable
   - $R_P(a, l)$, otherwise
5. If $R_P(b, l')$, $\exists e \in E_P$ such that $\mathit{src}(e) = l$, $l \in L_{P_i}, l' \in L_{P_j}$ and $i \neq j$, such that $a \in \mathit{processSeq}(\Lambda(\mathit{update}(e)), \{b\})$ then:
   - $\text{RelVP}_P(a, \text{channel}(e))$, if a is a value passing variable
   - $R_P(a, l)$, otherwise
6. If $\text{RelVP}_P(b, z)$, $\exists e \in E_P$ such that $\mathit{src}(e) = l$ and $\mathit{channel}(e) = z!$, $l \in L_{P_i}$, such that $a \in \mathit{processSeq}(\Lambda(\mathit{update}(e)), \{b\})$ then:
   - $\text{RelVP}_P(a, \text{channel}(e))$, if a is a value passing variable
   - $R_P(a, l)$, otherwise

Algorithm 10 processSeq(stmt_seq statements, variable relevant)

```
1: if (statements.isEmpty()) then
2:    return relevant
3: else
4:    relevant = processStat(statements.tail() , relevant)
5:    return processSeq(statements.withoutTail() , relevant)
6: end if
```

Algorithm 11 processStat(single_stmt stat , set⟨variables⟩ relevant)

```
1: if (stat ≡ u = expr) then
2:   if (u ∈ relevant) then
3:     removeVar(u, relevant)
4:     for all var ∈ used(expr) do
5:       insertVar(var, relevant)
6:     end for
7:   end if
8: else if (stat ≡ if(φ) B_1 else B_2) then
9:   for all var ∈ vars(φ) do
10:    insertVar(var, relevant)
11:  end for
12:  relevant = relevant ∪ processSeq(B_1), relevant)
13:  relevant = relevant ∪ processSeq(B_2), relevant)
14: else if (stat ≡ while(φ) B) then
15:  while (relevant ≠ temp) do
16:    temp = relevant
17:    for all var ∈ vars(condition) do
18:      insertVar(var, relevant)
19:    end for
20:    relevant = relevant ∪ processSeq(B, relevant)
21:  end while
22: end if
23: return relevant
```

Algorithm 12 insertVar(variable var, set⟨variable⟩ relevant)

```
1: if var.type == ARRAY then
2:   for (index_1 to index_n of var) do
3:     if (index_i != CONSTANT) then
4:       for all var ∈ vars(index_i) do
5:         relevant = insertVar(var, relevant)
6:       end for
7:       for (∀j ∈ range_i) do
8:         tempVar = var, with the i^th index replaced by j
9:         relevant = insertVar(tempVar, relevant)
10:      end for
11:    end if
12:    return relevant
13:  end for
14: end if
15: relevant.insert(var)
16: return relevant
```

Algorithm 13 removeVar(variable var, set⟨variable⟩ relevant)

```
1: bool isConstant = true
2: if var.type == ARRAY then
3:   for (index_1 to index_n of var) do
4:     if (index_i != CONSTANT) then
5:       isConstant = false
6:       for all var ∈ vars(index_i) do
7:         insertVar(var, relevant)
8:       end for
9:     end if
10:  end for
11: end if
12: if isConstant then
13:   relevant.remove(var)
14: end if
15: return relevant
```

Algorithm 14 propertyRelevance(Property prop)

```
markNodes(prop)
return getRelevantFromProperty(prop)
```

Algorithm 15 markNodes(Property prop)

if (prop $\equiv N \vee M$) **then**
  prop.trueLocs = N.trueLocs $\cup$ M.trueLocs
  prop.falseLocs = N.falseLocs $\cap$ M.falseLocs
else if (prop $\equiv N \wedge M$) **then**
  prop.trueLocs = N.trueLocs $\cap$ M.trueLocs
  prop.falseLocs = N.falseLocs $\cup$ M.falseLocs
else if (prop $\equiv \neg N$) **then**
  prop.trueLocs = N.falseLocs
  prop.falseLocs = N.trueLocs
else if (prop $\equiv l$) **then**
  prop.trueLocs = {l}
  prop.falseLocs = $L_{P_i} - \{l\}$, with L_{P_i} such that $l \in L_{P_i}$
else
  prop.trueLocs = {}
  prop.falseLocs = {}
end if

Algorithm 16 getRelevantFromProperty(Property prop)

if (prop $\equiv N \vee M$ or prop $\equiv N \wedge M$) **then**
  N.trueLocs = N.trueLocs $\cup$ prop.trueLocs
  M.trueLocs = M.trueLocs $\cup$ prop.trueLocs
  N.falseLocs = N.falseLocs $\cup$ prop.falseLocs
  M.falseLocs = M.falseLocs $\cup$ prop.falseLocs
  R = getRelevantFromProperty(N) $\cup$ getRelevantFromProperty(M)
else if (prop $\equiv \neg N$) **then**
  N.trueLocs = N.trueLocs = $\cup$ prop.trueLocs
  N.falseLocs = N.falseLocs = $\cup$ prop.falseLocs
  R =getRelevantFromProperty(N)
else if (prop $\equiv l$) **then**
  Nothing has to be done
else
  for (each $l \in L$) **do**
    if ($l \notin$ prop.trueLocs $\cup$ prop.falseLocs) **then**
      for (each $a \in$vars(prop)) **do**
        Add (a, l) to R
      end for
    end if
  end for
  return R
end if

CHAPTER 5

Transformation

Since we are now able to identify some irrelevant variables at a certain location we can use this information in the next step. This next step is about transforming the network of timed automata into another network of timed automata which has a, possibly, smaller number of states in the state space. We show that the transformed network is bisimilar to the original network and has at most the same number of states in the state space, but possibly a smaller number.
The definition of the transformation algorithm is divided into four stages. First we apply the same transformation as presented by Van de Pol and Timmer [30] by introducing a reset for every combination of (target) location and variable for which it holds that the variable is irrelevant at that location. And we prove that this transformation preserves bisimilarity and that it has the same or a smaller number of states. The second step is reducing the number of introduced resets through smart placement of the resets. However this step introduces, possibly, some extra states due to assignments to non-relevant variables that have been reset. In the third step we 'neutralise' these unnecessary assignments by undoing these assignments. The final step of the transformation takes care of resetting value passing variables at the end of a synchronisation and is based on section 4.3.2.

5.1 The basic transformation

Analogue to the definitions in [30] we define a transformation based on the non-relevancy of variables. Before we give this definition we first present an auxiliary definition concerning the initial valuation of variables.

Definition 21 (**Initial valuation of variables**)**.** Given a variable $a \in A$ we use init_a to denote the initial valuation of the variable a. If no initial value is supplied than $\text{init}_a = 0$.

Definition 22 (**Transformation**)**.** Given a network P of simple timed automata we have a transformed network P' of timed automata with each timed automaton $P'_i \in P'$ only differing from their original counterpart $P_i \in P$ in their edge-function: $P'_i = \{L_i, l_i^0, V_i, C_i, \Sigma, F_i, E'_i, I_i\}$. The difference of E'_i in

comparison with E_i is that the update statements ($update'(e)$) are changed according to the following definition

1. $ext\_update(e) = \{a = init_a \mid \exists i \in \{1 \dots n\} \cdot e \in E_i \wedge a \in local(P_i) \wedge \neg R(a, target(e)\}$
2. $update'(e) = update(e); ext\_update(e)$

Note that $update'(e)$ only deviates from $update(e)$ for variables that are irrelevant after taking e as we make clear using the following lemma.

We use $P@s$ and $P'@s$ to refer to a state s derived from the set of states of the underlying TLTS of a network P or its transform P'.

Lemma 23. For every $P_i \in P, e \in E_{P_i}$ and every state s, given that $guard(e)(s) =$ true it holds that $s.take(e) \cong s.take(e')$.

Proof. To show that $s.take(e) \cong s.take(e')$ we need to show that for all variables a such that $Relevant(a, s.take(e))$ we have $val_{s.take(e)}(a) = val_{s.take(e')}(a)$.

- If a is a local variable of the timed automaton P_i we have, by definition 9, $R(a, (target(e))$. Because of $R(a, (target(e))$ the condition $\neg R(a, target(e)$ evaluates to false and the set ext_update is empty for the variable a and $val_{s.take(e)}(a) = val_{s.take(e')}(a)$
- If a is a global variable the condition $a \in local(P_i)$ evaluates to false and the set ext_update is empty for the variable a and $val_{s.take(e)}(a) = val_{s.take(e')}(a)$

□

Based on this lemma we show that $P@s$ and $P'@s'$ are bisimilar, by first proving an even stronger statement.

Theorem 24. Let $\simeq$ be defined by:

$$P@s \simeq P'@s' \iff s \cong s'$$

then $\simeq$ is a strong bisimulation. The relation $\cong$ is used as it was defined for P

Proof. Let s_0 and s'_0 be states such that $P@s_0 \simeq P'@s'_0$, so $s_0 \cong s'_0$. We assume that $P@s_0 \xrightarrow{x} P@s_1$ and we have to prove that there exists a transition $P'@s'_0 \xrightarrow{x} P'@s'_1$.
By theorem 16 there exists a state s''_1 such that $P@s'_0 \xrightarrow{x} P@s''_1$ and $s_1 \cong s''_1$.
Secondly, by the operational semantics of UPPAAL, we have an edge e with guard$(e)(s'_0) =$ true, channel$(e)(s'_0) = x$ and s'_0.take$(e) = s''_1$.
By definition 22 we have $P'@s'_0 \xrightarrow{x} P'@s'_0$.take$(e')$
By lemma 23 s'_0.take$(e) \cong s'_0$.take(e').
Now by transitivity and reflexivity of $\cong$ (Lemma 11), we get $s_1 \cong s''_1 = s'_0$.take$(e) \cong s'_0$take(e'), hence $P@s_1 \simeq P'@s'_0$take(e'). By symmetry of $\cong$ this completes the proof.

□

Corollary 25. Let P be a network of simple timed automata, P' its transform, and s a state. Then, $P@s$ is strongly bisimilar to $P'@s$.

We now show that our choice of $update'(e)$ ensures that the state space of P' is at most as large as the state space of P. We first prove the invariant that if a variable is not relevant for a state that the value of the variable is equal to its initial value.

Proposition 26. For the network P of timed automata invariably $(\neg \mathit{Relevant}(a, s) \wedge \neg \mathit{isGlobal}(a)) \Rightarrow \mathit{val}_s(a) = \text{init}_a$

Proof. This proof is trivial, because this is exactly what is done in definition 22. □

Using this invariant we can now prove the following lemma, providing a functional strong bisimulation relating the states of $P@\mathit{init}$ and $P'@\mathit{init}$

Lemma 27. Let h be a function over states, given for any s by

$$h_a(s) = \begin{cases} \mathit{val}_s(a) & \text{if Relevant}(a, s) \vee \text{isGlobal}(a), \\ \mathit{init}_a & \text{otherwise.} \end{cases}$$

then h is a strong bisimulation relating the states of $P@\mathit{init}$ and $P'@\mathit{init}$

Proof. Let s_0 and s'_0 be states such that $h(s_0) = s'_0$. Also assume that $P@s_0 \xrightarrow{x} P@s_1$. We show that there exists a transition $P'@s'_0 \xrightarrow{x} P'@s'_1$ such that $h(s_1) = s'_1$ (the proof of the opposite direction is completely symmetric).
By definition of h it follows that $s_0 \cong s'_0$, so by Lemma 23 and Theorem 24 there is a s''_1 such that $P'@s'_0 \xrightarrow{x} P'@s''_1$ and $s_1 \cong s''_1$.

- Assuming that for an arbitrary variable a it holds that $\mathit{Relevant}(a, s_1) \vee \text{isGlobal}(a)$, this implies that $\mathit{val}_{s_1}(a) = \mathit{val}_{s''_1}(a)$, so by definition of h we obtain $h_a(v_1) = \mathit{val}_{s_1}(a) = \mathit{val}_{s''_1}(a)$.
- Assuming that $\neg \mathit{Relevant}(a, s_1)$ and $\neg\text{isGlobal}(a)$, we have by Lemma 10 and symmetry $\neg \mathit{Relevant}(a, s''_1)$, so by Proposition 26 it follows that $\mathit{val}_{s''_1}(a) = \mathit{init}_a$, so by definition of h we obtain $h_a(s_1) = \mathit{init}_a = \mathit{val}_{s''_1}(a)$. In conclusion, for all a in all cases we have $h_a(s_1) = \mathit{val}_{s''_1}(a)$, so $h_a(s_1) = s''_1$.

□

Because this relation between P and P' is a function, and the image of every function is at most as large as its domain, the following corollary is immediate.

Corollary 28. The number of reachable states in P' is at most as large as the number of reachable states in P.

Note that in theory the number of states is infinite due to the delay steps which can take every value in $\mathbb{R}_{\geq 0}$, however we assume the use of clock zones in UPPAAL as described (for instance) in [1] making the number of reachable states finite.

5.2 Minimising the number of resets

With the results of the previous sections we can indeed reduce the number of states in the state space, however it also introduces a reset assignment on every edge for each variable that is not relevant at the destination of the edge. One can imagine that the number of reset assignments can become quite large and as a result these numerous resets can make the model a lot harder to read after the transformation. Therefore it is better to only reset variables at points where they are not already equal to the initial value. For instance, if a variable is not relevant in two consecutive locations it is of no use to introduce two resets instead of only one at the first incoming edge. Therefore, our first adaptation is to only reset irrelevant variables at the beginning of a path of locations where that variable is not relevant. Essentially this means that the definition remains almost the same, making only a slight adaptation in the definition of $ext\_update(e)$, by requiring that the variable, that is going to be reset, is relevant at the source of the edge. By minimising the number of resets we improve on the work of [30]. In [16] Garavel and Serwe present an algorithm that also tries to minimise the resets but they require an additional pass of the automata.

Definition 29 (**First attempt of an improved transformation**)**.** Given a network P of timed automata we have a transformed network P' of timed automata with each timed automaton $P_i' \in P'$ only differing from their original counterpart $P_i \in P$ in their edge-function: $P_i' = \{L_i, l_i^0, V_i, C_i, \Sigma, F_i, E_i', I_i\}$. The difference of E_i' is that $update'(e)$ is changed, resulting in the following

1. $ext\_update(e) = \{a = \text{init}_a \mid e \in E_i \wedge a \in local(P_i) \wedge R(a, src(e)) \wedge \neg R(a, target(e)\}$
2. $update'(e) = update(e); ext\_update(e)$

In order to prove that this reduction in the number of resets does not increase again the number of states we have to show that the invariant introduced in Proposition 26 still holds. However this is not the case and to show this we give a counterexample. For instance, in figure 5.1 ($\text{init}_x = 0$), the algorithm inserts a reset after the use of x in $x < 5$. However the dead assignment $x = 3$ causes that x can have both the values 0 and 3 at locations l_3, l_4 and l_0, creating an unnecessary larger state space (in comparison with the original algorithm). The conclusion is that this first attempt failed and that we have to look for another solution which we consider in the next section.

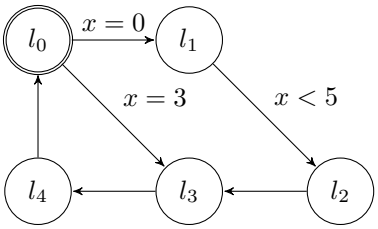


Figure 5.1: Example showing minimal number of resets is not sufficient

5.3 Improved transformation

The problem we are facing is that the invariant of Proposition 26 no longer holds and that we have a minimal amount of resets but also a possible increase in the number of states, sometimes even more than the state space of the original network of timed automata. As shown in the example of Figure 5.1 assignments to variables that are already not relevant can cause problems. There are two possible solutions to this problem:

1. Completely remove the assignments to non-relevant variables.
2. Introduce extra resets on each edge where non-relevant variables are changed.

The problem with the first option is that a variable can be irrelevant at both source and destination but still the assignment to that variable can be relevant, Figure 5.2 shows this. Consider the case that variable a is relevant at l_2. As a consequence b will be marked relevant after processing the statement $a = \text{test}[b])$ and again marked not relevant after processing the statement $b = c + 2$. Now b is not relevant at both the source and destination of the edge, but we can not completely remove the assignment because the assignment has influence on the behaviour of the program. Therefore we choose the second option and introduce additional resets to cope with the assignments to non relevant variables.
We could achieve the first option by applying substitution of the variable b in the second statement or combine both options by a more detailed analysis but we decided not to do this. A negative consequence of this is that we would end up with a more complex model that differs more from the original model, therefore making it harder for the developers to compare both models. While, if we only introduce resets (as shown in Figure 5.3), the original model is left untouched and only additional assignments are introduced. This makes it easier for the developers to understand what is going on, while achieving the same decreased state space.

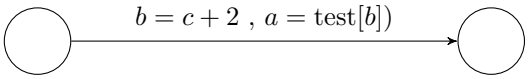


Figure 5.2: Assignments to non-relevant variables can be relevant

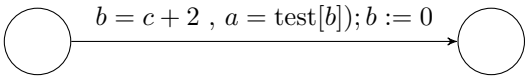


Figure 5.3: Assignments to non-relevant variables can be relevant

Definition 30 (**Improved Transformation**)**.** Given a network P of simple timed automata we have a transformed network P' of simple timed automata with each timed automaton $P'_i \in P'$ only differing from their original counterpart $P_i \in P$ in their edge-function: $P'_i = \{L_i, l_i^0, V_i, C_i, \Sigma, F_i, E'_i, I_i\}$. The difference of E'_i is that $update'(e)$ is changed resulting in the following:

1. $ext\_update(e) = \{a = \text{init}_a \mid e \in E_i \wedge a \in local(P_i) \wedge \neg R(a, target(e)) \wedge (R(a, src(e)) \vee a \in changed(e))\}$
2. $update'(e) = update(e); ext\_update(e)$

Lemma 31. Let P be a network of simple timed automata, P' its transform according to Definition 22 and P'' the transform of P according to Definition 5.3 then the number of reachable states of P' and P'' is the same.

Proof. To show that this improved transformation has the same reduced state space as the first transformation, Definition 22, it is sufficient to prove that the invariant of Proposition 26 $((\neg Relevant(a, s) \wedge \neg isGlobal(a)) \Rightarrow val_s(a) = \text{init}_a)$ is true for all reachable states.

For the initial state of the corresponding timed labelled transition system $val_s(a) = \text{init}_a$ trivially holds. We now assume it holds for an arbitrary state s and we show that it holds for all states s' which can be reached from s by taking an edge e. We distinguish the following possibilities:

- If $Relevant(a, s')$ or $isGlobal(a)$ the invariant is always true.
- If $\neg Relevant(a, s') \wedge \neg isGlobal(a)$ we have two possibilities: a can be relevant or not at state s:
  - $Relevant(a, s)$ - For this case the condition of the definition of $ext\_update(e)$ always holds and therefore by definition the invariant holds.
  - $\neg Relevant(a, s)$ - For this case we again get two possibilities: a is changed or not on edge e.
    - $a \in changed(e)$ - Also for this case the condition of the definition of $ext\_update(e)$ always holds and therefore by definition the invariant holds.
    - $a \notin changed(e)$ - If the value of a is not changed we have $val_s(a) = val_{s'}(a)$ and because of the assumption that the invariant holds for state s we have $val_{s'}(a) = \text{init}_a$

□

5.4 Resetting value passing variables

After we have defined the set of special global variables, value passing variables, in section 4.3.2 we now extend the transformation process with a final transformation in order to gain benefit from these value passing variables.

Definition 32 (**Final transformation including resets of value passing variables**)**.** Given a network P of simple timed automata we have a transformed network P' of simple timed automata with each timed automaton $P'_i \in P'$ only differing from their original counterpart $P_i \in P$ in their edge-function: $P'_i = \{L_i, l_i^0, V_i, C_i, \Sigma, F_i, E'_i, I_i\}$. The difference of E'_i is that $update'(e)$ is changed resulting in the following:

1. $ext\_update(e) = \{a = \text{init}_a \mid e \in E_i \wedge a \in local(P_i) \wedge \neg R(a, target(e)) \wedge (R(a, src(e)) \vee a \in changed(e))\}$

2. $vp\_update(e) = \{a = \text{init}_a \mid a \in \text{VPvars} \wedge a \in \text{used}(e)\}$
3. $update'(e) = update(e); ext\_update(e); vp\_update(e)$

Notice that ext_update and shared_update do not overlap, because $a \in \text{local}(P_i)$ and $a \in$ sharedVars are never true at the same time, as global variables are the only ones that can be shared variables.
Therefore proposition 26 still holds because for every local variable the proposition is invariably true. In order to reason about this final transformation we claim another proposition which ensures that every shared variable always has its initial value (only during the processing of an edge it can have other values).

Proposition 33. For every state $s \in S$ of the network P of timed automata invariably ($a \in$ sharedVars $\Rightarrow val_s(a) = \text{init}_a$

Proof. We assume $a \in$ sharedVars, otherwise the invariant is trivially true. Initially val $= \text{init}_a$. By definition 4.3.2 we know that the only edges on which a is changed are sending edges. Also that on each corresponding receiving edge $a \in \text{used}(e)$ and that there are no other edges for which this holds.We also know from the semantics of UPPAAL that the update part of the sending edge is always executed before the update part of the receiving edge. Definition 32 ensures that after each use of a shared variable a the value of a is reset to its initial value and we can conclude that after each change of the value of a it also is reset before the new state is calculated ensuring val $= \text{init}_a$. □

The above shows us, for the final version of the transformation, that the proposition still holds ensuring that each relevant variable has its original value and each irrelevant variable has its initial value. The last part is to give an updated version of lemma 27 to show that a similar functional bisimulation also holds for the final version of the transformation.

Lemma 34. Let h be a function over states, given for any s by

$$h_a(s) = \begin{cases} val_s(a) & \text{if Relevant}(a, s) \vee (\text{isGlobal}(a) \wedge \neg\text{isSharedVariable}(a)), \\ init_a & \text{otherwise.} \end{cases}$$

then h is a strong bisimulation relating the states of $P@init$ and $P'@init$

Proof. Let s_0 and s'_0 be states such that $h(s_0) = s'_0$. Also assume that $P@s_0 \xrightarrow{x} P@s_1$. We show that there exists a transition $P'@s'_0 \xrightarrow{x} P'@s'_1$ such that $h(s_1) = s'_1$ (the proof of the opposite direction is completely symmetric).
By definition of h it follows that $s_0 \cong s'_0$, so by Lemma 23 and Theorem 24 there is a s''_1 such that $P'@s'_0 \xrightarrow{x} P'@s''_1$ and $s_1 \cong s''_1$.

- Assuming that for an arbitrary variable a it holds that $Relevant(a, s_1)$, this implies that $val_{s_1}(a) = val_{s''_1}(a)$, so by definition of h we obtain $h_a(s_1) = val_{s_1}(a) = val_{s''_1}(a)$.
- Assuming that isGlobal$(a) \wedge a \notin$ Sharedvars the proof is analogue to the proof above.
- Assuming that isGlobal$(a) \wedge a \in$ sharedVars, we have by proposition 33 val $= \text{init}_a$.

- Assuming that $\neg Relevant(a, s_1)$ and $\neg$isGlobal(a), we have by Lemma 10 and symmetry $\neg Relevant(a, s_1'')$, so by Proposition 26 it follows that $val_{s_1''}(a) = init_a$, so by definition of h we obtain $h_a(s_1) = init_a = val_{s_1''}(a)$. In conclusion, for all a in all cases we have $h_a(s_1) = val_{s_1''}(a)$, so $h_a(s_1) = s_1''$.

□

CHAPTER 6

Implementation

After we have looked, in the previous chapters, at the theory of UPPAAL, our relevance algorithm and the accompanying transformation algorithm we now take a more practical look by looking at the implementation part of this thesis. Before we start with implementing the algorithms from the previous sections we first take a closer look at the practical side of the tool UPPAAL and the accompanying C++ library UTAP.

6.1 Theory versus Practice

One of the things with theory versus practice is that in the theory everything can look perfect and complete but when you try to implement something, you always will encounter differences between the theory and the practice making things a lot worse. In this research we have the same problems, first because of all the 'extra' features of UPPAAL, secondly because the language is even richer then the language of the theory, making it harder to cope with all the different possibilities. Finally we have to deal with how everything is working in UPPAAL and in the library UTAP. Because we are not free to choose how everything is organised inside the library we sometimes have to cope with some restrictions. Before proceeding with the actual implementation we first present an overview of all the 'special' features of the tool UPPAAL and the library UTAP.

6.1.1 Templates & Process instantiation

A network of timed automata of the theory is represented in UPPAAL as a *system* of *processes*. In UPPAAL you define an automaton in a template by drawing the graphical structure of the automaton and along that declare the accompanying imperative code. By instantiating this template you get a process. There are however some more options to create a process from a template which we present below.

Multiple process instantiations from a template Instead of instantiating a single process from a template it is also possible to instantiate multiple pro-

cesses from the same template. Each of these processes has the same graphical structure, the same function declarations and the same set of variables (these variables are still local and there are different versions of the variable for each process). The only problem is that in the end we want to introduce the resets without messing up the .xml file with a complete different structure. Therefore we have chosen to analysis templates instead of the processes, as the templates are the structures that are defined in the tool. In the case of multiple instantiations of one template we mark this template in order to indicate that some kind of resets are not possible for this template.
For example, if we have a system of multiple processes which are all instantiated from a single template it would, during the analysis, look like we are dealing with one process. A global variable will be treated in the same way as a local variable as there is only one process. However in the case of one process we can also reset non-relevant global variables, however in this example, because of the multiple template instantiations it is not possible and therefore we have to check, when resetting global variables, if we are dealing with a multiply-instantiated template.

Global & local variables In section 4.1.2 we defined which variables we consider global variables and which variables we consider to be local variables. In UPPAAL this distinction can be derived directly from the specification of the models. This because each template has its own section to declare variables and those variables are local variables of that template. Besides that there is a separate place to declare variables which can be accessed by every template and those variables are the global variables.

Template parameters A template definition can have parameters, which have the same syntax as C++, being that a call-by-reference parameter should have an ampersand in front and call-by-value parameters are not prefixed with an ampersand. The only difference with the semantics of C++ is that array variables that are parameters should be prefixed with an ampersand to be passed by reference. Also it is required for both clock and channel parameters to be call-by-reference, therefore they should always be prefixed with an ampersand.

In the case of a call-by-reference parameter the local parameter should be substituted by the global variable whenever the local parameter is used. We do not explicitly make this substitution but we keep track of every parameter/variable combination. Whenever we encounter a local parameter we look up the corresponding global variable.

6.1.2 Select statement

Until now an edge consists of three elements: guards, channels and updates. In UPPAAL there is another type of edge which is called *selections*. Selections are the first of the elements that are executed when an edge is taken and gives us the possibility to, non-deterministically, bind a value from a given range to a identifier. The scope of this identifier is limited to the edge it belongs to and therefore it can only be used in the other three elements of that edge. Because an identifier from a selection can only be used and not changed we can treat

this identifier as being a constant value with the consequence that the impact on our algorithm is limited.

6.2 Implementation of our tool

In the coming sections we describe how we implemented our relevance algorithm including the transformations that are made. We describe the whole process of the prototype tool from input to processing and finally how we write the result back in such a way that UPPAAL can cope with the result. Note that the whole process acts independently of UPPAAL itself (but uses the UTAP library), which made it a bit less complex as it was not necessary to (completely) understand and explore the implementation of UPPAAL itself.

6.2.1 Input

The input of our tool is the same as the input UPPAAL uses, namely an .xml file for the specification of the model and a .q file for a specification of the queries of the requirement properties of the model. There are files with other file-extensions and formats (of earlier versions of UPPAAL) that can be opened in UPPAAL but in this thesis we only take a look at the file format that is used in the current version of UPPAAL.

The .xml model file The models that are modelled in UPPAAL are saved as .xml files according to the Document Type Definition (DTD) of UPPAAL, which can be found at http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd. To access the relevant information of the models defined in the .xml files we could write our own parser for the .xml files. However the developers of UPPAAL provide also a C++ library, Uppaal Timed Automata Parser Library (UTAP). Using the UTAP library we can transform a model, specified in a .xml file, into a **TimedAutomataSystem**. Using this TimedAutomataSystem we can, in section 6.2.2, easily access all the relevant parts of the model specification without having to write an own parser.

The .q requirements file The .q file, containing the requirements of the model that needs to be verified, is a textual file containing for each query both the query itself and (possibly) an accompanying description. Both are written to the file exactly the same as specified in the verifier part of UPPAAL. However we are only interested in the first part and using **UTAP::PropertyBuilder** and **ParseProperty** we get a list of properties with for each property the property expressed as an **UTAP::expression_t**. This list of expressions will serve as an input to algorithms 7 and 8 for declaring variables relevant due to use in properties, as expressed in section 4.3.3.

6.2.2 Analysing

Having a parsed version of both the specifications, the model and the requirements, we can apply our algorithms of sections 4 and 5. These algorithms can be divided into several subtasks which we describe separately. The whole process of analysing the models can be divided into the following tasks:

- Determining the type of variable: global, local, shared, parameter or template.
- Determining the relevant variables:
  - Due to use in guards, channels or invariants (clause 1 and 2 of definition 7)
  - Due to use in a property (clause 3 of definition 7)
  - Due to transitions, possibly containing assignments (clause 4, 5 and 6 of definition 7)
- Transforming the model, applying the resets

In the following sections we describe each of the subtasks above:

Determining the type of variable

In the theory of section 4 we have divided the set of variables into three disjoint sets, being local, global or value passing variables. The practice adds one type of variable which is a parameter variable, a variable that is declared globally but transferred to a local process instantiation by using it as a parameter in the template instantiation.

Global variables We define a set⟨symbol_t⟩ globals in which we store the global variables obtained from the TimedAutomataSystem which has a **declaration_t** object with a field **variables**. Also, if a template is instantiated by multiple processes, we add the variables of that template to globals. Using the method isGlobal we can check whether a variable is a global variable.

Value passing variables A subset of the global variables are the value passing variables. For each global variable we check if it is a value passing variable in the method initShared(). This is done in two steps: We first iterate over all edges and if the edge is an receiving edge we iterate again over all edges for each used variable on this edge. In this new iteration we check that each used variable if on all corresponding sending edges (sending edges with the same channel as the receiving edge) the variable is changed. If that is the case then we have a possible value passing variable and we store the value passing variable in combination with the channel id.
The second step is that we check if the value passing variable is not used on any other edge. To check this we again iterate over all edges and if a variable is used on a non-receiving edge it is removed from the set of value passing variables. If the variable is used on an receiving edge we check if the corresponding channel corresponds with any of the stored channels for that variable. (A variable can be a value passing variable for multiple channels)

Local variables We do not explicitly define which variable is a local variable, because if we need to know if a variable is a local variable we just check if a variable is not a value passing, global or parameter variable, resulting in knowing if it is a local variable.

Parameter variables A template can have local parameters which are instantiated when a process is instantiated from this template. Every time we encounter this local parameter the variable is substituted with its actually use. To keep track of these mappings we use a set⟨pair⟨symbol_t, expression_t⟩⟩ to determine for each parameter variable (expression_t) with which local parameters of which processes it corresponds.

Determining the relevant variables

The next step is implementing the relevance algorithm as specified in section 4. The purpose of this algorithm is to mark variables relevant for specific locations. We specified a struct in order to represent these relevant 'combinations'. This struct, *relevant*, has three fields:

- var - The variable that is relevant given as an **expression_t**
- state - The location for which the variable is relevant, given as a **symbol_t**
- process - The corresponding template of the automation, given as a integer (index in the set of templates of the TimedAutomataSystem)

We could have excluded the process integer from the struct definition and determine it based on the location, because a location uniquely belongs to a template. However, by explicitly storing the template integer we save ourselves from unnecessary iterating over all templates every time we need to look up the process of a location.
The choice to store relevant variables as a **expression_t** was necessary in order to store array variables without making it too complex. Normally a variable is represented by UTAP as a symbol_t, however, if we call getSymbol() on an expression that contains an array variable we only get the identifier part of the array variable but not the array indices. We solved this by 'manually' parsing expressions and retrieve the variables ourselves and store them as expression_t objects instead of symbol_t objects.

Relevancy of variables in guards, invariants or channels According to clause 1 and 2 of algorithm 7 a variable is relevant at the source of an edge if it used in the guard or channel expressions of that edge and it is also relevant at a location if it is used in the invariant expression of that location. By iterating over all edges of all templates we can get both the guard and the channel expressions from respectively the fields **edge_t.guard** and **edge_t.sync**. Secondly by iterating over all states of all templates we get the invariant expressions by accessing the field **state_t.invariant**.
The next step is to retrieve all the variables from these guard, channel or invariant expressions. The UTAP library gives us an
expression_t.collectPossibleReads() method only this method is not sufficient for our needs because of the already mentioned complexity with array variables, therefore we defined an own *getReads(expression_t)* method. This getReads() parses an expression_t object and gives a set of (sub)expressions containing all the variables, array variables and function calls. And by recursively calling this getReads() method for indices of array variables and channel

expressions we also get the variables and function calls that are used as indices for both types.
While iterating over these resulting set of expressions we make for each variable or array variable a relevant item and call for this relevant item the method *insertRelevant()*, which inserts it correctly into the set of relevant items.

Relevance of variables in properties Clause 3 of the relevance algorithm is about marking variables relevant that are used in one of the properties specified in the .q file. Instead of marking all variables, that are used in property, relevant at all locations we improved on this by the algorithms of section 4.3.3. As input for these algorithms we have the set of property expressions from section 6.2.1. The implementation is divided into three parts, which we each present in the following paragraphs. These three parts are the algorithms 7 and 8 and the transformation of each of the property expressions into a parse tree that can be traversed by the two algorithms.
The first part is to construct the parse tree of a property. To construct the tree we first declare a **Node** object, which has the following properties:

- children - a set⟨Node⟩ containing the children of the Node
- expr - expression representing the sub tree of which this Node is the root
- trueLocs - a set⟨symbol_t⟩ of locations at which expr is always true
- falseLocs - a set⟨symbol_t⟩ of locations at which expr is always false
- type - an integer telling if this Node is a leaf or not, a location or not, negated or not

By parsing the property expression in **parsePropertyExpr** we build up a tree of Node's with subtrees representing sub-properties of a LEADSTO, OR, AND or negated property and leafs being either an expression containing variables or a location symbol.
Secondly we implement algorithm 7 in the method **markNode()**, which walks over the tree of Node's and is almost a direct copy of the algorithm described differing only in the addition of the LEADSTO property. This is a property (R) specifying that whenever a property P holds eventually a property Q holds. However we can not say anything about R being always true or always false as the truth value of Q is not based solely on the 'current' location but on all eventually possible locations.
Finally algorithm 8 is implemented in the method **parsePropertyNode()** inserting all used variables into the set of relevant items for all locations minus the set of locations which are either false or true. Also, while traversing the tree, the sets trueLocs and falseLocs are copied to the corresponding set of it's children, because whenever a property is always true or always false the value of each sub-property is also not important.

Iterative relevance Having described the implementation of clause 1,2 and 3 of definition 7 we proceed with describing the implementation of the three remaining clauses, which mainly consist of the implementation of algorithm 7. Clauses 4,5 and 6 all deal with marking variables relevant at the source of an edge based on the relevance of some variable at the target of an edge. We do

not check this for each relevant variable separately but we build a set of relevant variables (via the method *getRel()*) that is given to algorithm 10 (the method *processEdge()*)which in turn gives the set of relevant variables at the source of the edge which we then insert into the set of relevant items.

The method *getRel(set⟨relevant⟩ * setRel, symbol_t src, int proc))*
This method gives us the set of relevant variables that are used as input for the method processEdge(). All three clauses 4,5 and 6 give us a condition for which to include a variable into the input set. We iterate (ix) over all relevant items (of the whole network) and include those variables that meet one of the following three conditions.

- Clause 4 - If a variable is relevant at the target of the edge it should be included. This is done by checking if the state of the relevant item equals the target of the edge and that both processes are equal:

  !(ix→state.getName().compare(src.getName())) && proc == ix→process

- Clause 5 - If a variable is relevant at some other process (global variables) than the variable should be included. This is done by checking if the variable of the relevant item is a global variable and that the process not equals the process of the edge:

  isGlobal(ix-⟩var.getSymbol()) && proc == ix→process

- Clause 6 - If the edge is a sending side and the variable is (at the receiving side) marked relevant for the channel of the edge is should be included. This is done by not iterating over all relevant items but over the set of all relevant value passing variables for a specific channel. If the edge is a sending edge and the channel of the edge corresponds to the channel of the relevant value passing variable then this variable is included:

  if(!six→second.toString().compare(eix→sync[0].toString()))

It is possible that the iterator points to a parameter variable. If that is the case we perform the same checks but before inserting the variable into the resulting set we first transform the parameter variable to its local identity

The method *processEdge(expression_t expr, set⟨expression_t⟩ * rel, int proc)* Based on set of variables determined by getRel() (parameter rel) this method returns a set of variables that are relevant before processing the (part of an) edge *expr*. There are four different kind of expressions possible: a comma expression (multiple expressions separated by a comma), an assignment, a function call or an inline-if construction.
A **comma expression** is always a combination of two expression separated by a comma. For instance, if you enter, in UPPAAL, a comma expression of three expressions, you will get a comma expression which has a first part

that is the first expression and a second part that is a comma expression of the second and third expression. However, for our implementation it does not matter of which expressions the comma expression consists. For both parts of the comma expression we recursively call the method processEdge(), first we call processEdge for the second part (because algorithm **??** also works from the end of an edge to the beginning of an edge). Secondly we call processEdge for the first part but this time we pass the resulting relevant set of the first recursive call as parameter.
If the expression is an **assignment expression** we first check if the value on the left hand side is relevant at the source of the edge. If the expression is relevant we mark all variables on the right hand side relevant. Secondly, regardless of if the left hand side is relevant, we process each function call on the right hand side, however if the left hand side is relevant we also mark the return value of the statements relevant and so we pass on the relevance to the inside of the function. It may seem not logical to process a function call if the left hand side is not relevant but even if the return value is not relevant there can be other assignments to relevant variables inside the function making changes to the set of relevant variables. How we cope with processing functions can be read further on.
Function calls not in the right hand side of an assignment but separately on an edge are treated in the same way as a function call in the right hand side of an assignment of which the left hand side is not relevant. Also for the exact processing of the function look at the next paragraph.
The last possibility is that there is an **inline-if construction** on the edge, which looks as follows: condition ? expr_1 : expr_2. We evaluate such an expression also from the end to the beginning, meaning we first evaluate both expressions (true and false) and combine those sets. If one of the expressions makes any changes to the set of relevant variables we also evaluate the conditional expression at the beginning.

Functions The statements of an function are not directly accessible in a similar way as a statement directly on an edge. They have to be accessed by an visitor. We implemented a class MyVisitor that implements the UTAP::StatementVisitor. For every possible statement, like if, while or return statement there is a method that needs to be implemented. Almost all of the implemented methods are trivial, but for the return statement we had to know if the return value is relevant or not in order to correctly pass on the relevance inside the function. Therefore we have a special flag in the reducer class that keeps track if, in the case of a function call, the return value is relevant or not.
One thing to consider is the fact that our algorithm works from the target of an edge to the source of an edge. For the body of the function (which is a block-statement) this means that we first have to evaluate the last statement of the body/block and that we end with the first statement of the block.
After we have processed the first statement of the body of a function we check if any of the parameters of the function is marked relevant and if so we replace the parameter by the corresponding argument of the function call.

Inserting a relevant item into the set of relevant items Every time a call is made to insertRelevant() (for relevant items) or insertRelExpr() (for

expressions, used in the function visitor) we can not directly insert the item or expression into the set of relevant variables. Of the several cases that are possible only the case that we are handling a normal variable does not require extra attention and can be directly inserted into the set of relevant variables. However if we are dealing with either a parameter variable, a dot expression or an array variable we have to do a bit of processing first.
In the case that we want to insert a **local parameter** we can not insert this local parameter but we have to insert the corresponding parameter variable as mentioned in section 6.2.2. Therefore for every variable that is inserted we first look if that variable is a local parameter or not and if so we replace it by the corresponding variable and call the method recursively for the updated variable. Therefore if in another process the same variable (with possibly a different local parameter name) is used we know that we are dealing with a relevant variable.
The second check we do is to check if the variable is a **dot-variable**. A dot-variable is a variable that is made of two parts connected by a dot. The second part is the actual variable, while the first part can either be an process identifier or a struct identifier.
Thirdly we check if a variable is an **array variable**. If we are dealing with an array variable we insert the variables according to algorithm **??**.
Finally, before inserting the variable, we check if the variable should be inserted into the set of relevant variables (at a location) or into the set of value passing variables (at a channel).

6.2.3 Output

At some point the resets have to be inserted in the model and the model needs to be written to a .xml such that it can be opened and viewed in UPPAAL. A solution for this would be to update the edge_t.assign field of the edges of the TimedAutomateSystem by adding an assignment $x = \text{init}_x$ for each variable that needs to be reset. Consecutively we would write the TimedAutomataSystem to a .xml file. The UTAP library does not directly support this and next to that we only need to introduce some resets. Therefore we decided to scan the original (input) .xml file ourselves and look for the correct locations in the .xml file to insert the resets.

The scanning process consists of two scans of the .xml file. The first scan is to determine the x and y coordinates of the graphical elements of the model in order to place the resets graphically in a logical place which makes the resulting model readable. The second scan looks for edges and for each edge checks if a variable should be reset on this edge and if this is the case then it adds the reset at the end of the update part of the edge.

6.3 Validation

In order to be sure that our tool works as it should be we have tested it on the demo models that are supplied alongside the UPPAAL distribution and of course also on the case studies that we present in the next chapter. The UPPAAL demo models can be found in every UPPAAL distribution and are the

following:

- 2doors.xml
- bridge.xml
- fischer.xml
- interrupt.xml
- train-gate.xml

All five models are processed and transformed correctly resulting in models that produce the same results from the supplied properties from the corresponding .q files, unfortunately no reductions were achieved for these models.
However there are some other constructs possible in UPPAAL such that our tool does not function 100% correctly for all possible constructs of UPPAAL. We now give a short overview of (some of) the remaining problems:

- It is possible in UPPAAL to specify on an edge (in the update statement) an inline-if statement $(\varphi ? B_1 : B_2)$, that executes B_1 if φ is true and otherwise B_2 is executed. A normal inline-if is processed correctly, however an inline-if can also be used in the right hand side of an assignment and that construction is not processed correctly at the moment
- We need to check for present resets before resetting variables. For example, if the original model already resets an irrelevant variable y to zero by $y = 0$ our tool still adds another reset, which is not necessary.
- At this moment we are not able to determine the initial value of variables other than normal variables (not for arrays/constructs and such variables). It has to be found out if these initial values can be retrieved through the library UTAP, otherwise an extra own parser for the declarations has to be made.
- The last found problem is that our tool cannot cope with array declarations that contain 'difficult' expressions. For instance if we need to mark an array variable relevant that is declared as int$A[N + 1]$. We are able to retrieve the value of N through UTAP, but we are not able to calculate the value of $N + 1$ in order to insert all variables $A[0] \dots A[N]$.

- The relevance of function calls and consequently the return statements is not completely perfect. If inside an function a second function call is made, which is possible, than the return value of the second function call is relevant or irrelevant based on the return value of the first function call and not based on the relevance of the second function call.

To conclude we show that our implementation performs according to the theory by recalling the example of section 5.2:

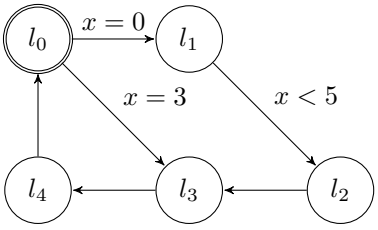


Figure 6.1: Example showing minimal number of resets is not sufficient

Our tool outputs (next to the transformed .xml model) the following:

```
Relevant: L1 - x - 1
B: In template 1 at edge L0 --> L3, variable - x - was reset to 0
A: In template 1 at edge L1 --> L2, variable - x - was reset to 0
```

CHAPTER 7

Case studies

The work presented in the previous sections, theory and implementation, may seem quite nice but the most important thing at the end is if the presented work is performing as expected. In order to show that our algorithm does what we claim it does we present three case studies, showing that reductions are achieved. An even more important part of these results is that we make it easier for modellers to specify their models by reducing the overhead of making sure the state space is minimal.
For each of the three case studies we start by providing some background information about the case study and presenting the UPPAAL models we are going to use, followed by the actual results.

7.1 Case 1: Handshake Register

In [30] Van de Pol and Timmer use a model of a handshake register of Hesselink [20] to evaluate their reduction algorithm for linear process equations. We have remodelled their models and our algorithm, as expected, can achieve similar reductions.

We first transform the Writer and Reader process (algorithms 17 and 18) to UPPAAL models (see figures 7.1 and 7.2). There are various aspects we have modelled slightly different than in the original models:

- We use the data set D to represent the $get(x)$ command.

- UPPAAL does not have the option to use *bits* as variables and booleans are also not an option as they can not be used in an array of channels. Therefore we model the bits as integers ranging from 0 to 1 and make use of modulo 2 to flip a bit.

Algorithm 17 The Writer

1: 0. get(x); $a := \neg B$;
2: 1.$Y.a.(\neg C.a) := x$;
3: 2.$C.a := \neg C.a$;
4: 3.$A := a$; goto 0

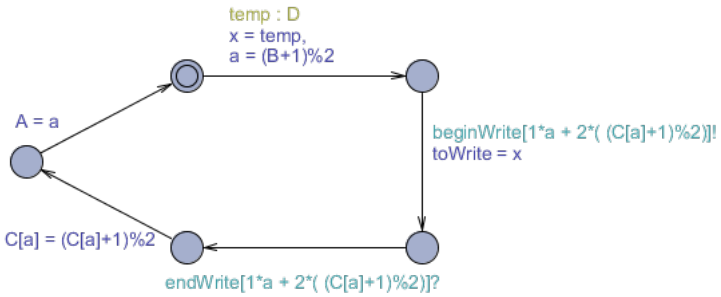


Figure 7.1: UPPAAL model of the Writer process

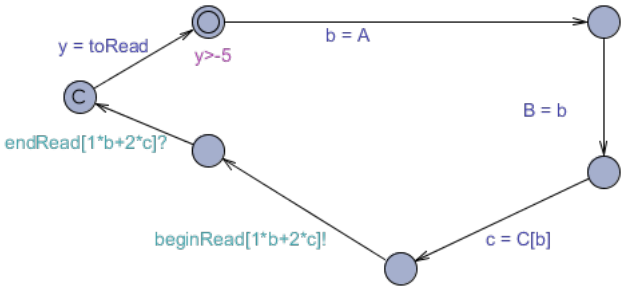


Figure 7.2: UPPAAL model of the Reader process

The final step of the remodelling is the linear process equation of the Registers, algorithm 19. The resulting model can be found in figure 7.3. This model is a bit larger than the Reader and Writer models, but can not be split into two distinct models. One could think this is possible looking at the linear process equation which looks like to have two different parts for both Read and Write. However the conditions in line 2 and 3 require that both models are combined into one model. Consequently this leads to a larger model because all interleavings should be captured in the model.

Algorithm 18 The Reader

1: 0.$b := A$;
2: 1.$B := b$;
3: 2.$c := C.b$;
4: 3.$y := Y.b.c$; put(y); goto 0

Algorithm 19 The Registers

$$
\begin{array}{lll}
Y(i\colon \mathrm{Bool}, j\colon \mathrm{Bool}, r\colon \{1,2\ ,3\}, w\colon \{1,2,3\}, v\colon D, vw\colon D, vr\colon D) & & \\
\quad r = 1 & \Rightarrow \mathrm{beginRead}(i,j) \cdot Y(i,j,2,w,v,vw,vr) & (1)\\
+ \quad r = 2 \wedge w = 1 & \Rightarrow \tau \cdot Y(i,j,3,w,v,vw,v) & (2)\\
+ \sum_{x\colon D} \quad r = 2 \wedge w \neq 1 & \Rightarrow \tau \cdot Y(i,j,3,w,v,vw,x) & (3)\\
+ \quad r = 3 & \Rightarrow \mathrm{endRead}(i,j,vr) \cdot Y(i,j,1,w,v,vw,vr) & (4)\\
+ \sum_{x\colon D} \quad w = 1 & \Rightarrow \mathrm{beginWrite}(i,j,x) \cdot Y(i,j,r,2,v,x,vr) & (5)\\
+ \quad w = 2 & \Rightarrow \tau \cdot Y(i,j,r,3,vw,vw,vr) & (6)\\
+ \quad w = 3 & \Rightarrow \mathrm{endWrite}(i,j) \cdot Y(i,j,r,1,vw,vw,vr) & (7)
\end{array}
$$

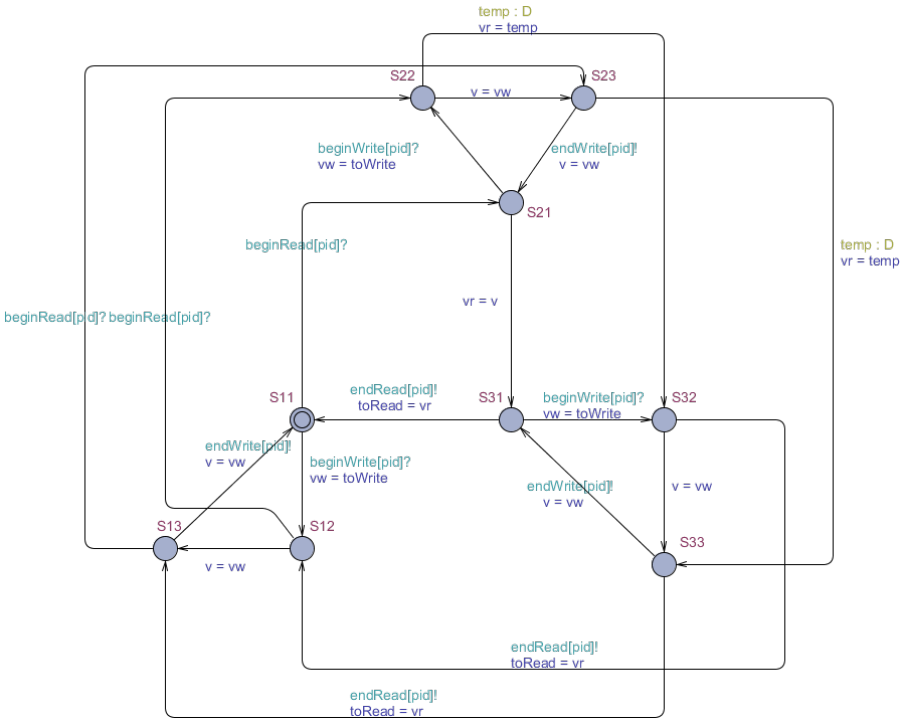


Figure 7.3: UPPAAL model of the Register process

7.1.1 Results

In order to calculate the total size of the state space we need to take care of two things:

- In the original linear process equation there are no properties supplied. However in UPPAAL we need a property that can be verified in order to run the verifier and get the total size of the state space.
- UPPAAL does not communicate with the environment. The get(x) (in-

put) method of the Writer is replaced by a random select, but we have no direct replacement for the put(x) method of the Reader.

To solve both problems we check the following property: $A\Box$ Reader.check imply $y >= 0$. This will mark the variable y relevant at location Reader.check and also cause the verifier to explore the total state space, making sure we get the size of the total state space. The relevant marking of the variable y is necessary because otherwise the whole process of reading/writing to the register would not be relevant and the only relevant variables would be the variables used in the indices of the synchronisation channels. If we run our tool on the UPPAAL models of the handshake register we get the following resets:

```
In template 1 at edge _id0 --> _id4, variable - a - was reset to  0
In template 1 at edge _id3 --> _id2, variable - x - was reset to  0

In template 2 at edge S33 --> S13, variable - vr - was reset to  0
In template 2 at edge S32 --> S12, variable - vr - was reset to  0
In template 2 at edge S13 --> S11, variable - vw - was reset to  0
In template 2 at edge S11 --> S12, variable - v - was reset to  0
In template 2 at edge S33 --> S31, variable - vw - was reset to  0
In template 2 at edge S31 --> S32, variable - v - was reset to  0
In template 2 at edge S23 --> S21, variable - vw - was reset to  0
In template 2 at edge S21 --> S22, variable - v - was reset to  0
In template 2 at edge S31 --> S11, variable - vr - was reset to  0

In template 3 at edge _id15 --> _id14, variable - b - was reset to  0
In template 3 at edge _id15 --> _id14, variable - c - was reset to  0
In template 3 at edge check --> _id18, variable - y - was reset to 0
```

The next step is to verify both models against the same property and the results can be found in Table 7.1. The number of visited states and stored states are the same and therefore we only give one value, which we compare to the number of states after transformation:

$\mid D \mid$	States before	States after	Percentage
2	526080	42624	8,102%
3	13655472	323232	0,170%
4	Out of Memory	1377280	
5	Out of Memory	4286880	
6	Out of Memory	10935552	

Table 7.1: Results of the handshake register case

Table 7.1 shows that a large reduction is achieved with a reduction that, manually, was not directly visible. Note that for 4,5 of 6 values of D this can not be generated without our reduction. This result is comparable with the result from

[30] and shows that the algorithm from their work can indeed also be applied to UPPAAL models.

7.2 Case 2: Root contention protocol

The second case study is a remodelled version of the 'Root Contention Protocol' case study performed by Simons and Stoelinga [28]. They used UPPAAL for the mechanical verification of the IEEE 1394 root contention protocol, which is an industrial leader election protocol with timing parameters playing an essential role. By using UPPAAL in combination with step wise abstraction Simons and Stoelinga investigate the timing constraints on the parameters, which are necessary and sufficient for correct protocol operation.

Simons and Stoelinga needed to model their version at a time UPPAAL did not support discrete variables but only supported clock variables. The original model consists of Node processes of which one needs to be selected as the leader and of Wire processes to one-directional connect two Nodes. Each Wire of the original process is a buffer with two places for messages, but because discrete variables were not available at that time each of the three possible messages is hard-coded into the model as can be seen in figure 7.4. By making use of data variables we can easily make the model clearer. In figure 7.5 the updated model can be seen. The actual behaviour of the 'Root Contention Protocol' can be found in the Node process (see figures 7.6 and 7.7). In this process very little changes have taken place between the version of Simons and Stoelinga and our version. We only needed to make sure that the Node process could communicate with the new UPPAAL model of the Wires. The templates are put together into the following system:

```
// Place template instantiations here.
Wire01 = Wire(0);
Wire10 = Wire(1);
Node0 = Node(0);
Node1 = Node(1);
// List one or more processes to be composed into a system.
system Wire01,Wire10,Node0,Node1;
```

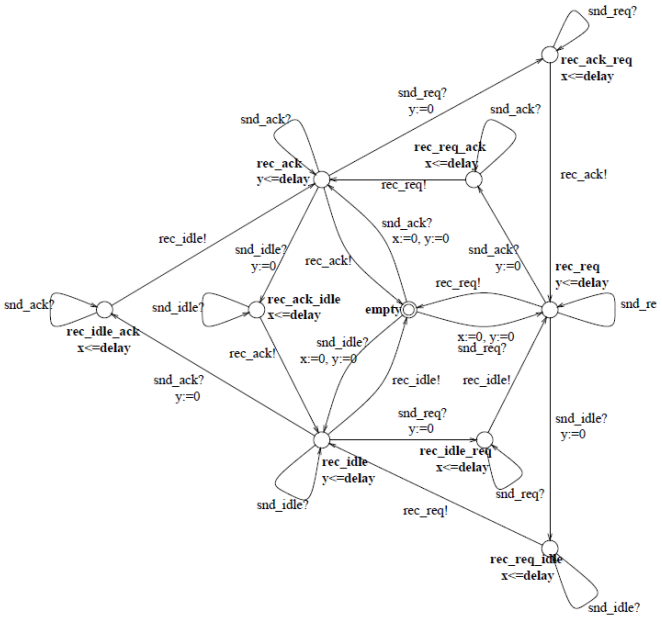


Figure 7.4: The old Wire model

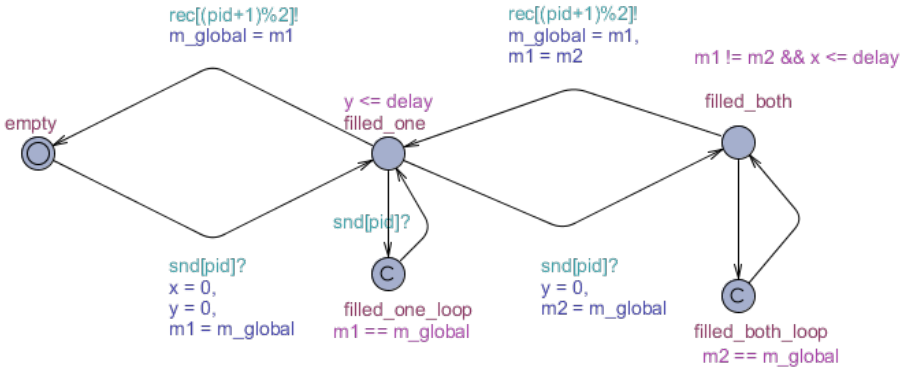


Figure 7.5: The new Wire model

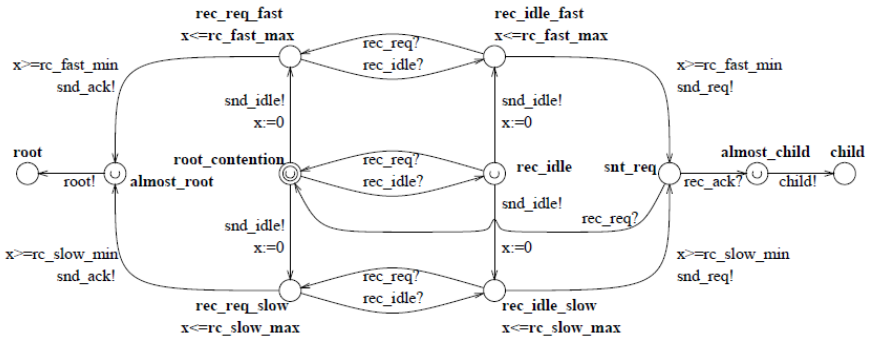


Figure 7.6: The old Node model

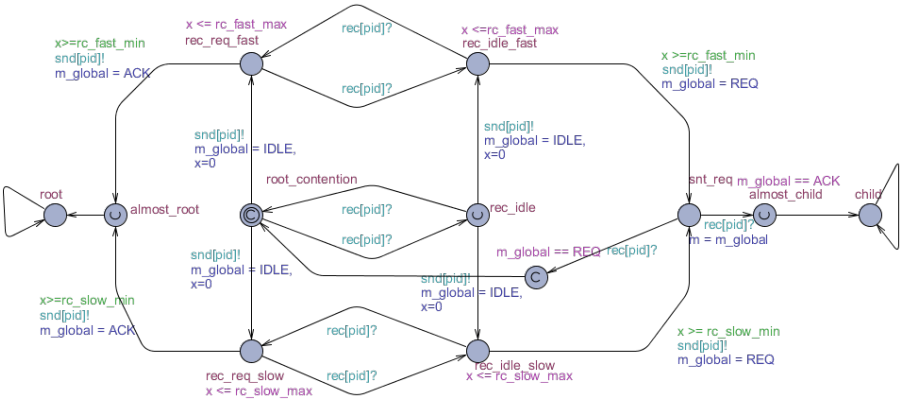


Figure 7.7: The new Node model

7.2.1 Results

Having modelled the UPPAAL models, as described in the previous section we now run our tool resulting in an transformed UPPAAL model with the following resets:

```
In template 1 at edge filled_both --> filled_one, variable - m2 - was reset to  0
In template 1 at edge filled_one --> empty, variable - m1 - was reset to  0
In template 1 at edge filled_one --> empty, variable - x - was reset to  0
In template 1 at edge filled_one --> empty, variable - y - was reset to  0

In template 2 at edge rec_idle_slow --> snt_req, variable - x - was reset to  0
In template 2 at edge rec_idle_fast --> snt_req, variable - x - was reset to  0
In template 2 at edge rec_req_fast --> almost_root, variable - x - was reset to  0
In template 2 at edge rec_req_slow --> almost_root, variable - x - was reset to  0
```

In order to measure the size of the total state space we check the following property:

$$A\Box \text{ not deadlock}$$

By running the UPPAAL verifier on both the original and the transformed UPPAAL model we get the following numbers of the state space:

	States explored	States stored
Before	233	225
After	110	76

Table 7.2: Root contention protocol reduction property 1

It is clear that our tool achieves a reduction in the size of the state space. The second property that we evaluate checks if the state *filled_both* is ever reached, ensuring that at some point both buffer places are filled.

$$E\Diamond \text{ Wire01.filled\_both}$$

	States explored	States stored
Before	48	41
After	48	41

Table 7.3: Root contention protocol reduction property 2

These results show that the verifier does not have to visit the total state space to evaluate this property, however it also shows that our tool does not cause in increase in the number of states of the state space that have to be searched.

7.3 Case 3: Shortest Tree Protocol for Wireless Sensor Networks

While the previous two case studies did not incorporate the use of functions and function calls the work of Everse on a Shortest Tree Protocol (STP) for Wireless Sensor Networks (WSNs) [15] is just the opposite, containing only one node with two self-edges. Most of the functionality is 'moved' to the imperative code making the models an ideal candidate to show that our algorithm work on functions.

In his work Everse tries to answer the question whether formal verification is suitable for supporting the design of wireless sensor networks. To answer this question he looked at a routing protocol that attempts to build a Shortest Path Tree in a distributed way. To answer the question he modelled the protocol in three model checkers, UPPAAL, SPIN and PRISM and used these model checkers to check various properties. We take a look at one of these UPPAAL models and try to see if our reduction algorithm can find reductions for this model.
For our case study we take a look at the UPPAAL protocol V1 as described in the work of Everse. We present the UPPAAL model itself (Figure 7.8 along with the global (Listing 7.1) and local (Listing 7.2) declarations. For a exact description of the models we suggest to take a look at the descriptions in [15].

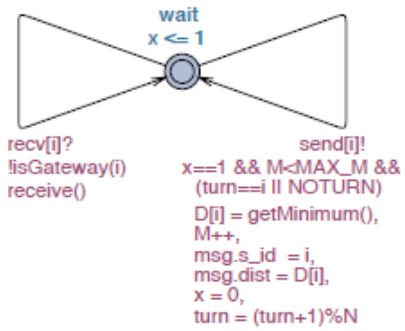


Figure 7.8: UPPAAL model of the shortest tree protocol

```
/****
Shortest Path Tree protocol for Wireless Sensor Networks
Author: W.M. Everse

Simple model:
- All models can hear each other with link quality of 100%
- Nodes operate synchronously
****/
// Number of nodes, including gateway(s) G
const int N = 4;

// Maximum number of message rounds
const int MAX_M = 10;

// Big number to represent 'infinite' distance
const int MAX_DIST = 10000;
```

```
17
18 // Define type Node_id, parameter of Node template
19 typedef int[0, N-1] Node_id;
20
21 // Synchronisation channel to model message sending/receiving
22 broadcast chan send;
23
24 // Model a message as a struct
25 // msg.s_id = sender id and msg.dist = distance
26 meta struct{
27     Node_id s_id;
28     int dist;
29 } msg;
30
31 // Determine whether the given node is a gateway node
32 bool isGateway(Node_id node) {
33     return node ==0;
34 }
```

Listing 7.1: Global declarations of Figure 7.8

```
1  // Node clock
2  clock x;
3
4  // Local round number
5  int M;
6
7  //Dist-to-G per node (D[y] = dist-to-G from y, perceived by this node)
8  int D[Node_id] = {MAX_DIST , MAX_DIST, MAX_DIST, MAX_DIST};
9
10 // Msg counters per node (R[y] = #messages received from node y)
11 int R[Node_i];
12
13 // The selected parent
14 meta Node_id parent;
15
16 // Function to determine the minimum distance
17 int getMinimum(){array}
18     meta int minval = MAX_DIST;          // To hold the min. found so far
19     meta int try;                        // To hold the next value
20     if( isGateway(i) ) return 0;         // Minimum dist-to-G of G is 0
21     if( M==0) return MAX_DIST;           // First round return MAX_DIST
22     for(j : Node_id){
23         if( R[j] > 0 && j != i && D[j] < MAX_DIST ){
24             try = M/R[j] + D[j];
25             if( (M % R[j]) >= (R[j] / 2) ) try++;  // Round to nearest Int
26             if( try <= minval){
27                 minval = try;
28                 parent = j;
29             }
30         }
31     }
32     return minval;                       // Return the min. found
33 }
34
35 // Function executed on message reception
36 void receive(){
37     R[msg.s_id]++;                   // Increase msg counter of sender
38     D[msg.S_id] = msg.dist;              // Update Dist-to-G of sender
```

Listing 7.2: Localdeclarations of Figure 7.8

As one can see in both listings UPPAAL model of Everse uses a lot of the 'extended' and/or imperative features of UPPAAL such as arrays, structs, function calls, for loops and if loops. This makes it an ideal case study to conclude that our tool can handle all these kind of features.

7.3.1 Results

The results of the third case study are a little bit complicated and it looks like no reductions can be achieved. However the cause of this is the *meta* keyword (line 26 of listing 7.8) of UPPAAL, which is described in the UPPAAL help-file as:

Integers, booleans, and arrays and records over integers and booleans can be marked as meta variables by prefixing the type with the keyword meta. Meta variables are stored in the state vector, but are semantically not considered part of the state. I.e. two states that only differ in meta variables are considered to be equal.

Since our tool marks the struct *msg* as a value passing variable there should be some reductions possible. Therefore we remove the meta keyword in front of the declaration of *msg*. This way we hope to show that our algorithm can also detect it, helping developers as they do not have to mark these variables meta manually any more. In Figure 7.4 one finds an overview of the size of the total state space for different number of nodes for the original model and the model with the meta keyword removed. We used the trivial property '$A\Box true$' to generate the total state space.

$\mid Nodes \mid$	With meta	Without meta
4	201	385
6	1233	3107
8	8601	26265
10	67233	240607
12	564201.	2335685

Table 7.4: Model of everse: total state space size

When we executed our tool it looked like we achieved some reduction because for the case of 4 nodes we got 268 states, which is larger than expected, and for the case of 6 nodes and more we get, strangely, smaller state space than with the meta keyword. The reason for this turned out to be that our tool does not work correctly for broadcast channels, therefore these numbers were not correct.

We examined the model more closely and it turned out that the channel used in the model is not a normal channel but a broadcast channel. A broadcast

channel is an one-to-many channel in which the sender synchronises with all corresponding receivers that are enabled. If no receiving edge is enabled then the sending edge is still enabled (many can be zero). The complications of a broadcast are that the receiving sides are executed in an arbitrary order preventing us to introduce resets at the end of a receiving side.

We can conclude that our tool, at the moment, does not reduce the state space for this case. However our tool does detect that the struct msg is a value passing variable. Future changes in our tool could include that we do not reset value passing variables but instead mark these variables as a meta variable resulting in a reduced state space (see figure 7.4).

CHAPTER 8

Conclusions

In this chapter we present the conclusions of our project. We first answer each of the sub-questions followed by the the answer to the research question itself. We conclude with some suggestions for further research.

Can the algorithm of [30] be translated to reduce the UPPAAL models by resetting local variables?

Sections 4 and 5 show that the algorithm can be translated without too many problems and by executing the same case study (section 7) as Van de Pol and Timmer used we show that our UPPAAL version of the algorithm can achieve similar results. Secondly our version also preserves the strong bisimulation of the original algorithm as proven in chapter 4. However due to the more complex nature of the UPPAAL language it was more than a simple translation as will be clear from the other subquestions.

Is there a way to reduce the state space by resetting global variables without constructing the total state space?

Due to the parallel nature of UPPAAL it is not possible, for us, to reset global variables because of all the possible interleavings between multiple processes. Because if a global variable is relevant at a process it is not possible to predict at which location all the other processes are. That is why a variable can seem irrelevant at one process but be relevant at one of the processes causing simply said that a global variable is relevant at (almost) all locations of all processes. However we have been able to identify a special kind of global variables, the value passing variables, that are solely used during synchronisation. We define these variables and incorporate them into our algorithm in section 4.3.2, consequently defining how to reset these variables in section 5.4. Finally, in the last case study, we suggest how to extend the definition of value passing variables to broadcast channels by marking these variables as *meta* variables

How can the algorithm be extended to include the state invariants of UPPAAL?

It turned out that the state invariants could easily be incorporated into the

original algorithm, translated from [30] to handle UPPAAL models. See clause 2 of definition 20. This clause shows that variables used in an invariant can be marked relevant in a way similar to variables used in a guard.

How can we incorporate the different features of UPPAAL into the algorithm?

This part of the research proved to be the most difficult as the language used in UPPAAL has been extended over the years by various constructs and features which we needed to incorporate into our design. We summarise how we handled these various structures:

- The **function calls** and the corresponding functions with their side effects were unfolded and thus eliminated by Definition 5 transforming a network of timed automata into a network of simple timed automata.
- Section 4.3.1 explains how we handle **array variables**,with non constant indices, in such a way that our algorithm also correctly marks these variables relevant.
- Instead of marking all variables used in a **property** relevant we improved this part of the algorithm in section 4.3.3 by 'predicting' the result of the evaluation of (sub)properties for certain locations.
- In chapter 6 we describe how we deal with the various features of the tool UPPAAL, like **structs**, **templates** and **template parameters**.

Is the tool beneficial for end users of UPPAAL, releasing them from optimizing the model for efficient verification?

The tool is definitely beneficial for users of UPPAAL because it offers the users an option to automatically insert resets at the correct places. This way the users have to worry less about achieving a minimal state space as the tool already takes care of it. The buffer used in case study 2 about the root contention protocol demonstrates this. One could suggest that a developer also could manually reset buffer places after getting the value stored in the buffer. However, using our tool, the developers no longer need to do this manually, which is always error prone, and can concentrate on different things.

What can control flow analysis applied to UPPAAL models achieve?

We have shown that our tool can handle the complicated language of UPPAAL models and can achieve reductions in various cases. It also turned out, during our research that most of the models used in various papers do not benefit from our tool. The main reason for this that, until now, developers, most of the time, applied the described resets manually. This manual resetting of variables by developers leads to the main benefit of our research being that we have shown that our algorithm and tool can simplify the modelling process for developers.

8.1 Future work

Although the algorithm looks finished there are some more things that can be considered to improve the algorithm such as the combination with other known techniques or some small improvements. We now describe these suggestions for future research.

Expression substitution In section we presented two possible solutions to the problem of assignments to irrelevant variables. One of the options was to remove these assignments completely, however this was not possible because a variable can be irrelevant at both source and target of an edge but this does not mean that a variable is than automatically irrelevant at the point of the assignment. Therefore we chose for the second option to introduce extra resets for these assignments making sure irrelevant variables always (at a location) have its initial value.

However if we take a look again at Figure 5.2 it seems that it is beneficial to substitute the right hand side of the assignment to b at further uses of b and consequently removing the assignment to b resulting in the edge as shown in Figure 8.1.

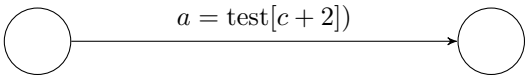


Figure 8.1: Substitution of assignments to irrelevant variables.

Combination with constant propagation In section 2.2.4 we we gave a short view on the compiler optimisation theory called constant propagation. It could be that by using constant propagation on UPPAAL models it turns out that for certain models certain branching conditions are either constantly true or false. Therefore some branches could prove to be unreachable making the use of the variables inside this branch irrelevant. The same technique technique can be used in combination with array indices. If we can 'predict' the value of an array index we will, when marking an array variable relevant, not have to mark the array variable relevant for all possible options of the range of an index. Both options can lead to a smaller set of relevant variables and possibly more resets.

Improve on the equivalence relation While the first two suggestions for future work were relatively concrete, the last one is more a small piece of brainstorming. Recall the equivalence relation $\cong$ as presented in section 4.2, which was defined as:

$$s \cong s' \Leftrightarrow \bar{l}_s = \bar{l}_{s'} \wedge \forall a \in A : (\text{Relevant}(a, s) \Rightarrow \text{val}_s(a) = \text{val}_{s'}(a)), \text{ where } s, s' \in S$$

The first part of the right hand side of this equation, $\bar{l}_s = \bar{l}_{s'}$, may offer possibilities for further study. Is this part required or can we make this requirement less strict making it possible to incorporate for instance symmetry.

Bibliography

[1] Rajeev Alur. Timed automata. In *Computer Aided Verification*, pages 688–688. Springer Berlin / Heidelberg, 1999.

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

[3] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. Uppaal: now, next, and future. pages 99–124, 2001.

[4] G. Behrmann, A. David, K.G. Larsen, J. Hakansson, P. Petterson, Wang Yi, and M. Hendriks. Uppaal 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126, Sept. 2006.

[5] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[6] Johan Bengtsson, W. O. David Griffioen, Kare J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using Uppaal. In *In CAV'96, LNCS 1102*, pages 244–256. Springer-Verlag, 1996.

[7] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *CONCUR '98: Proceedings of the 9th International Conference on Concurrency Theory*, pages 485–500, London, UK, 1998. Springer-Verlag.

[8] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *In Lecture Notes on Concurrency and Petri Nets*, Lecture Notes in Computer Science vol 3098. Springer–Verlag, 2004.

[9] Marius Bozga, Jean-Claude Fernandez, and Constantin Lucian Ghirvu. State Space Reduction based on Live Variables Analysis. In Gilberto

Cortesi, Agostino; File, editor, *Static Analysis 6th International Symposium, SAS'99, Venice, Italy, September 22-24, 1999, Proceedings Static Analysis 6th International Symposium, SAS'99*, volume 1694 of *Lecture Notes in Computer Science*, pages 164–178, Venice Italie, 09 1999. Springer.

[10] Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking: History, Achievements, Perspectives*, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[12] Ron Cytron, Andy Lowry, and F. Kenneth Zadeck. Code motion of control structures in high-level languages. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 70–85, New York, NY, USA, 1986. ACM.

[13] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. pages 208–219. Springer-Verlag, 1996.

[14] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium*, page 73, Washington, DC, USA, 1996. IEEE Computer Society.

[15] Wouter M. Everse. Modelling and verification of a shortest path tree protocol for wireless sensor networks : Towards a platform for formal verification experiments, 2009.

[16] Hubert Garavel and Wendelin Serwe. State space reduction for process algebra specifications. *Theoretical Computer Science*, 351(2):131 – 145, 2006. Algebraic Methodology and Software Technology.

[17] Klaus Havelund, Arne Skou, Kim Guldstrand Larsen, and Kristian Lund. Formal modeling and analysis of an audio/video protocol: An industrial case study using Uppaal, booktitle = In Proc. of the 18th IEEE Real-Time Systems Symposium, year = 1997, pages = 2–13, publisher = IEEE Computer Society Press.

[18] M. Hendriks and K. G. Larsen. Exact acceleration of real-time model checking. In E. Asarin, O. Maler, and S. Yovine, editors, *Workshop on Theory and Practice of Timed Systems (TPTS'02)*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2002.

[19] Martijn Hendriks. Enhancing Uppaal by exploiting symmetry. Technical Report NIII-R0208, University of Nijmegen, October 2002.

[20] Wim H. Hesselink. Invariants for the construction of a handshake register. *Information Processing Letters*, 68(4):173 – 177, 1998.

[21] C. Norris Ip and David L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9(1-2):41–75, 1996.

[22] Agata Janowska and Pawel Janowski. Slicing of timed automata with discrete data. *Fundam. Inf.*, 72(1-3):181–195, 2006.

[23] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: theory and practice. *ACM Trans. Program. Lang. Syst.*, 16(4):1117–1155, 1994.

[24] Kim Larsen and Wang Yi. Time abstracted bisimulation: Implicit specifications and decidability. 1994.

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

[26] Doron Peled. Ten years of partial order reduction. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 17–28, London, UK, 1998. Springer-Verlag.

[27] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM.

[28] David P. L. Simons and Mariëlle Stoelinga. Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k, 2000.

[29] C. Thrane and U. Sørensen. Slicing for Uppaal. *Student Paper, 2008 Annual IEEE Conference*, pages 1–5, 15-26 Feb. 2008.

[30] J. C. van de Pol and M. Timmer. State space reduction of linear processes using control flow reconstruction. In *ATVA 2009 - Automated Technology for Verification and Analysis. 7th International Symposium, Macao SAR, China*, volume 5799 of *Lecture Notes in Computer Science*, pages 54–68, Berlin, October 2009. Springer Verlag.

[31] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branc. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[32] Karen Yorav and Orna Grumberg. Static analysis for state-space reductions preserving temporal logics. *Form. Methods Syst. Des.*, 25(1):67–96, 2004.