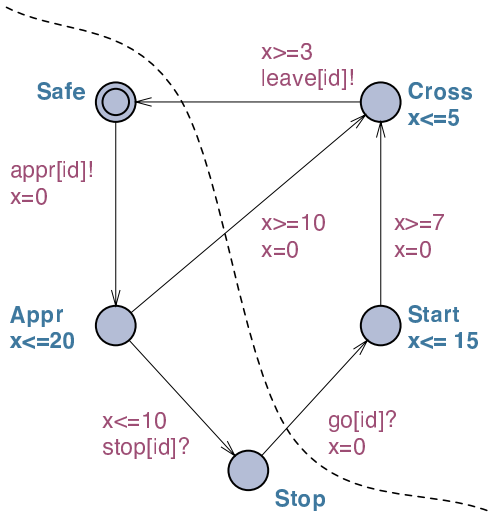
Slicing for UPPAAL

Uffe Sørensen and Claus Thrane



AALBORG UNIVERSITY
Department of Computer Science, June 2007

Title:

Slicing for Uppaal

Project Scope

Master Thesis
Feb 6 - June 15, 2007

Project Group

D603A

Group Members

Uffe Sørensen
Claus Thrane

Supervisor

Prof. Dr. Kim G. Larsen

Number of copies: 6

Number of pages: 93

Abstract

The focus of this thesis is to introduce *slicing* for Uppaal [9]. Slicing is a technique based on static analysis used to reduce the syntactic size of models or applications. In this thesis, we show how slicing may be used to construct reachability preserving reductions of Uppaal models possibly improving the performance of the tool. Using automated slicing in Uppaal will eliminate the need for users to manually optimize models for faster verification of a certain property. Moreover, it allows less experienced users of Uppaal, which unknowingly may design models, containing unnecessary large amounts of data, to verify properties which Uppaal otherwise would have been unable to check.

The slicing is done by analyzing the control-flow of a model, in order to extract dependencies between its components, which is then stored in a data structure known as a *system dependency graph* (SDG). Computing the relevant components (the slicing set) of the model is then achieved using graph analysis on the SDG. The sliced model is then constructed using only the components included in the slicing set.

In order to formally define and prove correctness of the slicing approach, we define an extended notion of the timed automata formalism [3] which constitutes a non-trivial subset of the modeling language used in Uppaal. Moreover, to stress the complexity of verifying properties for Uppaal models and further motivate the use of slicing, the general problem of model-checking is described and known theory is presented.

Finally, a prototype implementation of slicing for Uppaal has been developed to show that the slicing approach presented can in-fact be extended to the complete language of Uppaal. The prototype has been used to conduct a set of experiments, in which we succeeded in finding a design flaw in a model provided by students from the department of communication technology at AAU. Moreover, the experiments indicate that slicing does indeed provide an effective and beneficial component for Uppaal, which leads us to encourage the further development of the prototype towards inclusion in the offical distributed version of Uppaal.

On the attached CD is a digital version of this thesis along with source code and a pre-compiled binary of our prototype. The contents of the CD and any updates can also be found at http://www.cs.aau.dk/~crt/utasa/

Preface

Creating software specifications is an extremely debated subject in the software industry. Some argue that it is actually impossible to describe the requirements of a (non-trivial) system prior to the system implementation and in some cases even after. This raises interesting questions about the correctness of systems. How are we supposed to test or verify that the final product is what we wanted, when we do not know what that was in the first place?

Although we might not be able to define all the properties of a system, we are often able to get a pretty good idea about most of them, and in some cases we are even capable of developing a formal specification of these. The question which then arises is how to insure that these properties are fulfilled by the given system. One approach, and probably the most used, is *testing*. Massive amount of research have been dedicated to the subject of testing software systems. Testing is an essential part of the software development process for all types of systems, but for some purposes testing does not suffice. Leaving aside superficial functional requirements, like the visual design and feel, there are basic properties which are always desirable in all systems. Among these are safety properties, which insure that the system never deadlocks or livelocks. But there are also other properties which we are able to define for individual systems, such as time requirements of real-time systems or time sensitive applications, or more general things, such as: The system must always react to certain types of commands e.g. shutdown. This is where *software verification* is necessary. Whereas testing may show the presence of errors it can not help us guarantee the correctness of the systems' behavior. Unlike testing, software verification is not widely deployed in mainstream industry, this is primarily because verification is often a very daunting task, and the presence of supporting tools is minimal.

The idea behind the work presented in this thesis is founded in the fact that we see an increasing need for such tools. Since work on the tool Uppaal already has been very successful at Aalborg University, it was natural to study the area of model-based verification using Uppaal and timed automata, to find way to bring traditional developers closer to the world of verification. Although Uppaal already is very efficient, the consequences of state-space explosion are not immediately natural to developers of software systems. Our goal has therefore been to develop a technique allowing developers to model their systems (for the purpose of verification) closer to the way that software systems are developed. As software developers, we often decorate our core functionality with contextual data e.g. a protocol implementation would include functionality to monitor package loss and so on. The consequence may easily be that the states-space grows to a level where verification may consume too much time to be of practical value during the development process. In this thesis we present a technique known as slicing, which may be used to automatically determine the parts of Uppaal models which can influence the outcome of verifying a given property. Using this technique as a preprocessing tool to reduce the input model before verification, should ensure that the irrelevant functionality no longer presents a problem, leaving engineers free to model their systems, without thinking about state-space explosions.

Contents

Chapter

1 Introduction

In this Chapter an overview of the thesis is presented, including the contributions and the focus of the project. We present the motivation for software verification in general, followed by the specific motivation for the slicing technique introduced in this thesis. Finally, we present related work.

Contents

1.1 Software Verification and Model Checking

Proving the correctness of software in general is an extremely difficult task, and not surprisingly, it is in its most general form undecidable (Rice 1953[1]). Although verification remains highly complex, research has nontheless produced results which have enabled us to perform verification, with reasonable performance. This is achieved by exploiting domain specific knowledge, using over-approximations and abstraction techniques.

Software verification may be divided into two categories, the first is the *deductive* verification method, in which verification is based on properties in a mathematical theory. These properties are then proven or refuted using *theorem provers*. The second category is the *model-based* verification method, in which a model representation of the system under consideration is used in a variety of techniques such as simulation and state exploration.

Calculi and Logics

Calculi and logics are widely used in the area of verification. Classical calculi like CCS [43], TCCS [30] and CSP [33] have been used to model sequential and concurrent systems for many years. Since verification is very complex, we often use simplified models of our system, expressed in calculi. Many different calculi have been developed for the purpose of modeling different domains, examples include CCS [43], Mobile Ambient Calculus [17], π-Calculus and secure π-Calculus [49], which are used to model, for example standard reactive systems, mobile systems, protocols and secure protocols respectively.

Likewise, a considerable amount of logic frameworks have been developed to express properties of systems. Such logics may contain domain specific modalities such as in temporal logics *Computational Three Logic* (CTL) [23] or *Linary Temporal Logic* (LTL) [42] which include $\Diamond$ and $\Box$ modalities, others include modalities for space ($\text{✧}, \text{⌑}$) and so forth.

Software Verification

The term software verification traditionally applies to techniques and tools, used in the verification of a given system's implementation. Several tools have been developed with the intention of verifying correctness of systems implemented in classical imperative languages such as C and modern object oriented languages such as Java.

The SLAM toolkit[6], developed by Microsoft Research, statically analyzes a C program to check safety properties. The toolkit has two unique features; it does not require the programmer to annotate the source program (invariants are inferred); it minimizes noise (false error messages) through counterexample-driven refinement. Given a safety property to check on a C program P, the SLAM process [4] iteratively refines a boolean program abstraction of P using three tools:

[1] Rice's theorem: Any nontrivial property about the language recognized by a Turing machine is undecidable.

- C2BP, a predicate abstraction tool that abstracts P into a boolean program $\mathcal{BP}(P, E)$ with respect to a set of predicates E over P.
- BEBOP, a tool for model checking boolean programs.
- NEWTON, a tool that discovers additional predicates to refine the boolean program, by analyzing the feasibility of paths in the C program.

Like SLAM, BLAST [32] (**B**erkeley **L**azy **A**bstraction **S**oftware Verification **T**ool) is a verification system for checking safety properties of C programs. BLAST implements an abstract-check-refine loop to check for reachability of a specified label in the program. The abstraction made by BLAST is an on-the-fly predicate abstraction, known as *lazy abstraction* [32]. If there is no path to the specified error label, BLAST reports that the system is safe. If there is a path and the path is feasible, BLAST outputs the path as an error trace, otherwise, it uses the infeasibility of the path to refine the abstract model.

The Bandera tool set [27] is an integrated collection of program analysis, transformation, and visualization components designed to facilitate experimentation with model-checking Java source code. Bandera takes as input Java source code and a software requirement formalized in Bandera's temporal specification language, and it generates a program model and specification in the input language of one of several existing model-checking tools (including SPIN [34], dSPIN [36], SMV [39], and JPF [29]). Both program slicing (which is introduced below and later in Chapter 4) and user extensible abstract interpretation components are applied to customize the program model to the property being checked. When a model-checker produces an error trace, Bandera renders the error trace at the source code level and allows the user to step through the code along the path of the trace.

Model-Checking

In model-based software verification two approaches for proving correctness are traditionally considered. *Model Checking*, which given a model and a requirement specified in some modal logic, checks the property by exploring the structure of the model or its state-space based on the semantics of the system. Alternatively, *Specification checking* is based on a notion of equality. The model of the system in question and a simplified version (which is easily verified by hand) is checked with respect to equality e.g. *bisimilarity*.

Several model-checking tools are used in the industry today, the most prominent, to our knowledge, being SPIN [34]. The SPIN model-checker has evolved for more than fifteen years and is by now a very mature tool. Input models, expressed as automata, are defined in Promela (**Pro**cess **me**ta **la**nguage), which supports modeling of asynchronous distributed algorithms as non-deterministic automata. Properties to be verified are expressed as LTL formulae. In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user.

In contrast to SPIN, which does not support modeling of time, the model-checking tool Uppaal has been developed for the purpose of verifying real-time systems

modelled as a network of timed automata. UPPAAL is developed by the Department of Information Technology at Uppsala University in Sweden in collaboration with the Department of Computer Science at Aalborg University in Denmark. The input language for UPPAAL is a combination of a graphical representation of timed automata and imperative declarations of functions expressed in an extended subset of C. In order to verify properties of models, UPPAAL requires that such properties are specified in CTL.

Like UPPAAL, KRONOS [20] and its successor IF [25] are model-checkers, though based on the timed automata formalism using the TCTL [1] logic as property specification language. It can decide whether some property, expressed by a TCTL formula, holds for a timed automaton. Starting from a system consisting of several components, KRONOS computes the automaton corresponding to the synchronized product. KRONOS is one of the few tools which implements a model-checking algorithm for a timed temporal logic. For this reason, it allows one to verify liveness properties.

Since model-based verification does not attempt to verify properties of a system's actual implementation, it is of paramount importance to stress that verification using model-based techniques is only as good as the model of the system. The technique employed to produce the model on which verification is based, be this manuel or automatic generation, is crucial when using the verification result to reason about the actual implementation. Furthermore, exhaustive model-checking suffers from the so-called *state explosion*, which expresses the problem of exponential growth in the number of states for a model. Model checking timed automata which is the formalism used in KRONOS and UPPAAL is even known to be P-SPACE complete.

Combining Verification and Model-Checking

Although we have already classified UPPAAL as a model-checking tool, the fact that later versions of the tool support a non-trivial subset of C, should merit a classification as a hybrid. Although UPPAAL is already very efficient, the combination of imperative code verification and model checking timed automata is a very complex and time consuming task. The work presented in this thesis is inspired by the fact that work on UPPAAL primarily has been done at the timed automata level, optimizing and analyzing issues imposed by parallel composition of timed automata. In the following, we give a further introduction to the language used in UPPAAL, in order to motivate research into static analysis of the imperative language components and to achieve even better performance of the tool.

1.2 UPPAAL – Models with Imperative Code

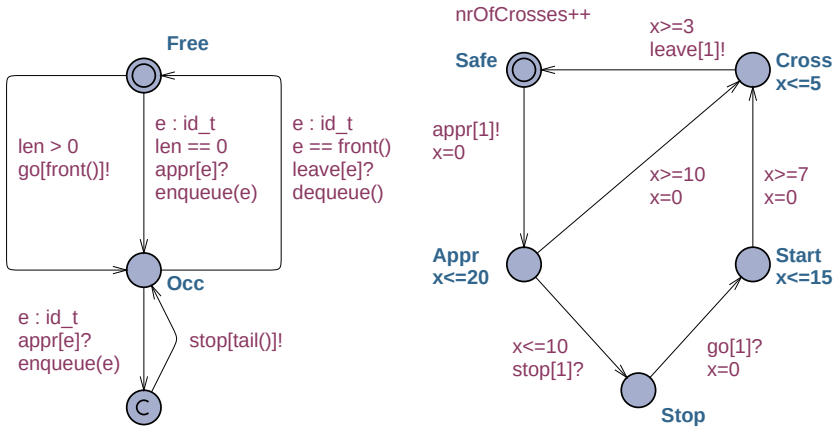
As of UPPAAL version 4.0, the modeling language has been extended with the possibility to add imperative code to models. Although the core language is still based on models expressed graphically, functions expressed in imperative C-like code may be called from the edges of the automata. Introduction of the imperative code has meant that users may easily express reusable complex operations as well as maintain the state of data in integer, array and struct types, which they were previously

required to "code" using the structure of the automata. Although the modeling task has become significantly easier for the user, verifying properties of the models now requires much more of UPPAAL. The imperative code is an integrated part of the model and hence must be dealt with by the model-checker.

The fact that imperative code is easily added to the models, encourages users to add peripheral variables to debug or monitor the behavior of the model under development. Although not critical to the models behavior, these variables will participate in the state space, putting even more load on the verification engine. In the following example we introduce a slightly modified version of a classical UPPAAL demonstration model, where variables have been added to simulate simple statistical data. We use this example to motivate the work done here and to illustrate how slicing may be used as a preprocess to remove these auxiliary variables. In Chapter 6, we introduce a real life example model designed by students at Aalborg University.

1.2.1 The Train-Gate Example

The following is a well known example demonstrating the features of UPPAAL called the *Train-Gate Example*. It models a train crossing problem, where a number of *trains* requests access to a crossing point in order to continue on their respective routes. The crossing point is governed by a *gate* which each train must signal to gain crossing authorization. Figure 1.1 illustrates the timed automaton modeling the *gate* and one of the *trains*.



(a) The gate

(b) The first of 6 train instances

Figure 1.1: Illustrates the timed automata in the *Train-Gate Example*

The original model shiped with UPPAAL contains 6 trains (instances), all of which are initially safe and a *gate* which is free. The *trains* concurrently signal their approach (**appr!**) and the *gate* will acknowledge the signal (**appr?**). If the "crossing" is free, it will let the train enter, otherwise it will immediately signal the train to stop (**stop!**). Having stopped a *train*, the *gate* will add the *train* to a queue, which

it will begin emptying whenever the "crossing" is free. Having achieved access, and successfully left the "crossing" the *train* will signal the *gate* of its departure (**leave!**). Whenever such a signal is received by the *gate*, it will signal the *train* in the front of the queue to go (**go!**). Having received the signal (**go?**) the *train* will enter the "crossing", signaling its departure when it has safely exited. Finally, whenever a *train* has exited, it will attempt to approach the *gate* again.

Declarations

Although the timed automata in Figure 1.1 contains the behavior just described, some of which may be obvious for some, the behavior induced by the decorated edges cannot be clear without introducing the imperative declarations in the model. Declarations in UPPAAL are either global or local to each timed automata instance, allowing locally and globally shared variables and functions. The global declaration in the *train-gate* model contains the following statements:

```
const int N = 6;         // # trains
typedef int[0,N-1] id_t;
chan        appr[N], stop[N], leave[N];
urgent chan go[N];
```

The statements respectively defines the number of *trains* (N) and an ID type (id_t) used it identify the *trains*. Next, three arrays of channels are defined, allowing communication between the *gate* and the *trains*. Each *train* communicates their approach and exit from the "crossing" outputting (written `!`) on the channel array `appr[ ]` and `leave[ ]` respectively, using their ID as index. Dually it will listen on (written `?`) the channel array `go[ ]` for its signal to enter the "crossing" and `stop[ ]` to stop. The *gate* will listen on all 6 channels simultaneously using a non-deterministically selected index "e". Using its local functions `front()` and `tail()` (described below) the *gate* signals the *trains* to stop or go, where the *train* signaled, depends on the ID returned by the functions.

Train Declarations

In order to "monitor" the number of times a single train successfully crosses the *gate*, each train has been decorated with a local integer variable `nrOfCrosses` which is incremented each time the train exits the "crossing".

```
clock x;
int nrOfCrosses;
```

Also each train has a local clock variable `x` which is used to model the time. In Figure 1.1(a) The predicate `x>=7` on the outgoing edge from start dictates a delay of at least 7 time units since the clock has been reset on the incoming edge. Having achieved an intuition of the *guards* we leave the formal definition of clocks and delays to be introduced in Chapter 2.

Gate Declarations

The declarations belonging to the *gate* is somewhat more comprehensive. We will assume that the reader is familiar with standard C-like imperative languages

used to express the functions and that no further introduction is required to read the syntax (in Section 2.3, we formally define a non-trivial subset of the UPPAAL imperative language seen here). Notice that the range declared for the integer variable `len`, restricts the domain of the variable to 7 values (N being 6); drastically reducing its contributions to the state-space of the model. Deviating from the original model shipped with UPPAAL, we have added the integer variable `numberOfTrainsWaitingToCross` which has no effect on the models behavior, but could help the user visualize the behavior in UPPAAL's simulator.

```
id_t list[N+1];
int[0,N] len;
int numberOfTrainsWaitingToCross;

void enqueue(id_t element) // Put an element at the end of the queue
{
    numberOfTrainsWaitingToCross++;
    list[len++] = element;
}

void dequeue() // Remove the front element of the queue
{
    int i = 0;

    if(numberOfTrainsWaitingToCross > 0)
    {
        len -= 1;

        while (i < len)
        {
            list[i] = list[i + 1]; i++;
        }
        list[i] = 0;
        numberOfTrainsWaitingToCross--;
    }
}

id_t front() // Returns the front element of the queue
{
   return list[0];
}

id_t tail() // Returns the last element of the queue
{
   return list[len - 1];
}
```

The intuition of the above code is that the *gate* will use the function `enqueue(..)` to register approaching *trains* in a queue, modeled by the array variable `list`, holding *train* IDs. The *gate* will use the `dequeue()` function to remove *trains* whenever they

signal `leave!`. Finally, the functions `front()` and `tail()` are used to get the first and last *train* ID in the queue.

1.3 Slicing UPPAAL Models

Although the *Train-Gate* example is very simple, it demonstrates very well the effects of introducing imperative code in UPPAAL. The fact that model-checking is highly dependent on the size of the state-space, quickly gets very clear to model designers when they begin to decorate the model with excess functionality. Although some users are aware of the state-space explosion problem, the effects of adding a single unbounded integer variable may be very surprising. An unbounded integer variable in UPPAAL is implemented as a 16bit signed integer, yielding alone 65536 states (2^{16}). The growth of the state-space for a single unbounded variable is then $|\textit{state-space}| \times 65536$, assuming $|\textit{state-space}|$ is its original size.

The focus of this thesis is to introduce a technique called *slicing*, which will allow users of the UPPAAL tool to add auxiliary data to their models, without affecting the state-space which is model-checked. Looking at the modified *train-gate* example above, it should be clear that the existence of the variable `nrOfCrosses` (which we have added to the *train*), is irrelevant. Under no circumstances can it affect any properties of the system, since the only reference to the variable is an update of itself (located on the outgoing edge from the *cross* vertex in Figure 1.1(b)). Although the variable `numberOfTrainsWaitingToCross` has no real purpose, with respect to the models behavior, its use in function `dequeue()` requires its presence, since the value of the expression `numberOfTrainsWaitingToCross > 0` controls the execution of critical update code. Were we to remove this (unnecessary) sanity check, we would once again realize that the variable could be removed.

Although the motivation given here in terms of slicing away auxiliary data, slicing would also be able to remove irrelevant components, with respect to a given CTL formula; that is, components which do not affect the specific behaviour which is to be verified.

1.3.1 Our Contribution - Automated Slicing

Using the intuitive notion of slicing, designers could manually remove excessive variables before verification is initiated, but they would then be required to reinsert them whenever an error is found or if they decide to extend the model.

Automating the slicing process may help model designers focus on the core properties of their models, without the concern of what an excessive use of e.g. debug information may do to the state-space. In this thesis we introduce a theoretical approach to compute slices for a non-trivial subset of the UPPAAL modeling language (referred to as *extendend timed automata* in Section 2.3) and present a proof of correctness. Furthermore, we present a prototype implementation, which is used to conduct experiments on a real life model.

1.4 The Structure of This Thesis

Having introduced the intuition behind program slicing and motivated its use for Uppaal, we continue in Chapter 2 to give a formal introduction to timed automata and the extended timed automata, which is the formalism used throughout the formal chapters of this thesis. Furthermore, in Chapter 2 we give the formal semantics of the extended timed automata. In Chapter 3 we discuss the general problem of model-checking and introduce the notion of regions, zones and the DBM data structure. Chapter 4 presents our approach of slicing the extended timed automata formalism, along with a formal proof of correctness. In Chapter 5 we introduce an implementation for slicing Uppaal models, using the discussed theory and in Chapter 6 we present experimental results. Finally, in Chapter 7 we present the conclusion and our final remarks. Furthermore, we present future work and the various interesting aspects and problems concerned with this work.

1.5 Related Work

Much work on slicing has previously been presented in the context of debugging or testing sequential or concurrent applications. Work on slicing has already been produced for a number of imperative languages. J. Hatcliff [28] shows how the traditional slicing concepts described by Wieser [52] and later S. Horwitz, T. Reps and D. Binkley [35] may be extended to slicing a more complicated model of multi-threaded Java programs with JVM concurrency primitives. Furthermore, he shows that a bisimulation-based notion may be used to determine correctness of slices. More closely related is the work by Janowska and Janowski [37], who show how slicing may be used for the formalism of *timed automata* defined by Alur and Dill [3]. Although both show that slicing is applicable for non-trivial languages, no one has (to our knowledge) ventured to show that slicing may be performed on a complex hybrid of imperative code and timed automata, which currently forms the basis of the modeling language used in Uppaal [40, 9].

Chapter

2 Timed Automata

This Chapter introduces a version of the timed automata formalism introduced in [3] extended with discrete variables and a small imperative language. The purpose of the extended definition is to show that slicing may be performed on timed automata resembling those used in the Uppaal modeling tool. The extensions made here, although comprehensive, are only a subset of the extensions introduced in Uppaal, which is referred to as an extended subset of C. Nevertheless, the features included are chosen such that it should be clear that the approach introduced in Chapter 4 may be extended to the complete language used in Uppaal.

In order to make the extensions apparent, we first introduce a simple definition which resembles the formalism originally defined in [3]. We then proceed to introduce the extended version.

Contents

2.1 Introduction to Timed Automata

Timed Automata is a formalism for modeling and verifying real-time systems. Based on the original work of Alur and Dill[3], several model checking tools have been developed using timed automata as the core of their respective modeling languages, examples include UPPAAL [9] and KRONOS [14]. Alternative formalisms, such as *Timed Petri Nets*, have also been researched with the same purpose in mind. Timed automata, as originally defined in [3], is based on finite state Büchi automata[1] extended with real-valued clocks. A simplified version of timed automata, called *Timed Safety Automata*, was introduced in [31] which allows specification of progress properties using local invariant conditions. This Section introduces timed safety automata and in the remainder of this thesis we will refer to timed safety automata simply as timed automata.

2.1.1 Timed Automata

Timed automata are fundamentally finite-state automata extended with a notion of time by adding real-valued clocks.
In addition, edges and locations may be decorated with predicates over these clocks, called clock constraints. Clock constraints are used to model forced and restricted progress in the automaton. Furthermore, the alphabet of timed automata denotes a set of channels which may be used for synchronization in networks of timed automata.

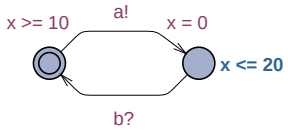


Figure 2.1: Example of a timed automaton

Figure 2.1 illustrates a trivial example of a timed automaton with two states and two edges. The automaton is willing to output on channel **a** followed by input on channel **b**. The edge leaving the initial state (indicated by an inner circle) is guarded by a predicate $x >= 10$, such that at least 10 time units must have passed before the edge is enabled, likewise "taking" the edge results in a reset of the clock x. Finally, the second location must be vacated within 20 time units dictated by the predicate $x <= 20$.

2.1.2 Clocks

Clocks are the central concept of timed automata. Initially, all clocks are zero and synchronously increased at the same rate. We use $\mathcal{C}$ to denote the set of clocks in an automaton.

[1] A Büchi automaton is an automaton designed to recognize (or generate) infinite words. The rule followed by a Büchi automaton is not to "end in an accepting state", but rather to traverse accepting states infinitely often in the course of its computation. See [16].

Clock Valuations

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

Clock Constraints

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

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

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

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

2.1.3 Synchronization

A notion of channels [43] is used to obtain synchronization between timed automata in a network (in parallel). Edges of timed automata are decorated with channels from the alphabet Σ (See definition 1). 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 is a channel in Σ. alternatively, we say it is willing to input if its edge is decorated with $a?$. Two 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 common set of channels Σ.

2.2 Basic Timed Automata

This Section introduces, for reason of comparison, the definition and the syntax and semantics of timed automata in line with what was introduced in [3], after which we extend the definition to match a non-trivial subset of the representation used as the modeling language for Uppaal.

Definition 1. *(Timed Automata)*
A *timed automaton* is a tuple $\langle L, l_0, \Sigma, \mathcal{C}, E, I \rangle$ where

- L is a finite set of locations
- $l_0 \in L$ is the initial location
- Σ is a finite set of channels
- $\mathcal{C}$ is a finite set of clocks
- $E \subseteq L \times \Psi(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times L$ is the set of edges
- $I : L \longrightarrow \Psi(\mathcal{C})$ 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 and l' are locations (source and target respectively), g is the set of clock constraints guarding the edge, a is the channel, which in some cases may be referred to as the action and r is the set of clocks which are reset.

Semantics

The semantics of timed automata is defined as a *timed labeled transition system* (TLTS) where states (or configurations) consist of a location and a clock valuation. Transitions are either delay transitions $\xrightarrow{d}$ (hence the TLTS) or action transitions $\xrightarrow{a}$, meaning that a system may either delay in the current location 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. $\mathcal{B}(\mathcal{C})$, we abuse the notation $\sigma \in I(l)$ 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 $\sigma \in I(l)$ and $(\sigma + d') \in I(l)$ 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. $\sigma \in g \wedge \sigma' = \sigma[r \mapsto 0] \wedge \sigma' \in I(l')$

Parallel composition

We use the term network to denote a model of parallel composed timed automata. A network of timed automata $\mathcal{A}$ is defined over a common set of clocks and channels and consists of n timed automata $\mathcal{A}_i = \{L_i, l_i^0, C, \Sigma, E_i, I_i\}$, where $1 \leq i \leq n$. A location in $\mathcal{A}$ is a *location vector* $\bar{l} = \{l_1, \ldots, l_n\}$ over locations for each $\mathcal{A}_i$. Updates to the location vector are written $\bar{l}[l_i'/l_i]$ to denote that automata $\mathcal{A}_i$ moves from location l to l'. 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 $\mathcal{A} = \langle L_i, l_i^0, \mathcal{C}, \Sigma, E_i, I_i \rangle$ be a parallel composition of timed automata $(\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n)$ 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 $\bar{l} = (l_1^0, \ldots, l_n^0)$. The semantics is defined in terms of a timed labeled 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 $\forall d' : \sigma + d' \in I(\bar{l})$ 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. $\sigma \in g, \sigma' = \sigma[r \mapsto 0]$ and $\sigma' \in I(\bar{l}[l_i'/l_i])$
- $\langle \bar{l}, \sigma \rangle \xrightarrow{\tau} \langle \bar{l}[l_i'/l_i, l_j'/l_j], \sigma' \rangle$ 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 $\sigma \in (g_i \wedge g_j), \sigma' = \sigma[r_i \cup r_j \mapsto 0]$ and $\sigma' \in I(\bar{l}[l_i'/l_i, l_j'/l_j])$

2.3 The Extended Timed Automata Formalism

The notion of timed automata introduced here is (as expected) based on the previous definition. 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. Also a notion of urgency has been added. We now say that locations may be either *urgent* or *committed*, in order to explicitly express that

delays are unacceptable. 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 times automata with discrete variables, we introduce a notion of variable valuations. A variable valuation is a total mapping $\omega : \mathcal{V} \cup retval \rightarrow \mathbb{Z}$ from a set of variables $\mathcal{V}$ to the set of integers. The variable $retval$ is a special variable which cannot be used for any other purpose than returning values from function calls (see Section 2.3.2 on $retval$). Finally, we use $\mathscr{D}$ to denote the set of all variable valuations.

2.3.1 Syntax of The Imperative Language

This Section introduces a non-trivial subset of the imperative language of UPPAAL, which we use throughout this thesis. The language presented here is chosen such that the correctness of our slicing 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.

Functions and Statements In order to manipulate discrete variables, we introduce the possibility of having functions (without recursion) which may be called in the update of discrete variables. We use f to denote a function and F to denote a set of functions defined in the following syntax:

```
funcDecl     ::= f(id_1, ..., id_n){stmt_seq}
stmt_seq     ::= ϵ | single_stmt stmt_seq
single_stmt  ::= if(φ){stmt_seq} | while(φ){stmt_seq}
                 | return expr | single_act
```

Where `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`. We use λ to range over statements in the production `single_stmt`. The syntax for function calls is defined as:

```
funCall  ::= f(expr_1, ..., expr_n)
```

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

```
expr ::= m | v | expr ⊗ expr | -expr | (expr) | funCall
```

By $Expr(\mathcal{V}, F)$, we denote the set of all possible arithmetic expressions over $\mathcal{V}$ and F.

The set of boolean expressions over discrete variables is defined in the production `bexp`, where `expr` $\in Expr(\mathcal{V}, F)$ and $\sim \in \{==, !=, <, >, <=, >=\}$.

$$\texttt{bexp ::= true} \mid \texttt{expr} \sim \texttt{expr} \mid \varphi \texttt{ \&\& } \varphi \mid \varphi \,||\, \varphi \mid \neg\varphi \mid (\varphi)$$

The set of all boolean expressions over $\mathcal{V}$ and F is denoted by $\Phi(\mathcal{V}, F)$ ranged over by φ.

Finally, actions over discrete variables $\mathcal{V}$ and functions F are defined by the production `single_act`, where $v \in \mathcal{V}$ and `expr` $\in Expr(\mathcal{V}, F)$. The set of all actions over $\mathcal{V}$ and F is denoted $Act(\mathcal{V}, F)$ and α denotes a sequence expressed in production `act_seq` over $Act(\mathcal{V}, F)$.

$$\begin{aligned}\texttt{single\_act ::= funCall} \mid v \texttt{ = expr} \mid \texttt{skip}\\ \texttt{act\_seq ::=}\ \epsilon \mid \texttt{single\_act act\_seq}\end{aligned}$$

Intuitively, α denotes a, possibly empty, sequence of discrete variable assignments and function calls (see updates in Definition 2.3).

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

$$\texttt{clockconst ::= true} \mid u \sim c \mid u_1 - u_2 \sim c \mid \psi \texttt{ \&\& } \psi$$

By $\Psi(\mathcal{C})$ we denote the set of all clock constraints over $\mathcal{C}$, ranged over by ψ. As with discrete variables, we use a production `single_asg` to define clock assignments, where $Asg(\mathcal{C})$, denotes the set of all assignments over $\mathcal{C}$ and β denotes a sequence of clock updates expressed in `asg_seq` over $Asg(\mathcal{C})$.

$$\begin{aligned}\texttt{single\_asg ::=}\ u = \texttt{expr} \mid \texttt{skip}\\ \texttt{asg\_seq ::=}\ \epsilon \mid \texttt{single\_asg asg\_seq}\end{aligned}$$

The basis for the distinction between the syntactic productions `single_stmt` and `single_act` may not be obvious at this point, hence a brief discussion is in order. In Section 2.3.3 we introduce an extended definition of edges and their updates. For this purpose, we restrict edge updates to a sequence of variable actions (α), in the `act_seq` syntax, since it enables the model designer to call functions containing more complex functionality. Moreover, variable actions may be used as statements within the definition of functions and are therefore also ranged over by λ.

2.3.2 Semantics of The Imperative Language

In this Section we present the semantics of the imperative language of the extended timed automata. For each transition in the transition system for timed automata networks (Definition 5), there may exist multiple transitions in the semantics given by the following operational semantics. Since the transition system of the timed automata network describes the level at which the execution of automata transitions may interleave, a big-step semantics is sufficient to describe the behavior of the imperative language. The semantics is given by the semantic catagories: *Arithmetic Expressions*, *Boolean Expressions*, *Statements* and *Function Calls*.

Arithmetic Expressions:

The function $[\![expr]\!]$ denotes the semantics of an expression $expr \in Expr(\mathcal{V}, F)$. $[\![expr]\!]$ is a mapping $\mathscr{C} \times \mathscr{D} \to \mathbb{Z} \times \mathscr{D} \times \mathscr{C}$. That is, it maps a pair of a clock and variable valuations into a value (integer) and a new clock and variable valuation. $[\![expr]\!](\sigma, \omega) = (val, \sigma', \omega')$ iff $\langle expr, \sigma, \omega\rangle \to \langle val, \sigma', \omega'\rangle$ wrt. the following operational semantics.

[constant] $$\frac{}{\langle m, \sigma, \omega\rangle \to \langle m, \sigma, \omega\rangle}$$ where m is a constant in $\mathbb{Z}$

[variable] $$\frac{\omega(v) = val}{\langle v, \sigma, \omega\rangle \to \langle val, \sigma, \omega\rangle}$$

[clock] $$\frac{\omega(u) = val}{\langle u, \sigma, \omega\rangle \to \langle val, \sigma, \omega\rangle}$$

Since the syntactic production `expr` allows function calls, the valuation of an expression may have a side effect, resulting in an updated variable valuation. (See page 28 on functions)

[binary-arit] $$\frac{\langle expr_1, \sigma, \omega\rangle \to \langle val_1, \sigma, \omega''\rangle \quad \langle expr_2, \sigma, \omega''\rangle \to \langle val_2, \sigma, \omega'\rangle}{\langle expr_1 \otimes expr_2, \sigma, \omega\rangle \to \langle val, \sigma, \omega'\rangle}$$ where $val = \lfloor val_1 \otimes val_2 \rfloor$ and $\otimes \in \{-, +, *, /\}$

Boolean Expressions:

Expressions $\psi \in \Psi(\mathcal{C})$ and $\varphi \in \Phi(\mathcal{V}, F)$ are boolean expressions over the set of clocks $\mathcal{C}$ and the set of variables $\mathcal{V}$ respectively. The semantics $[\![\psi]\!]$ of ψ and $[\![\varphi]\!]$ of φ are mappings $\mathscr{C} \times \mathscr{D} \to \{tt, ff\} \times \mathscr{C} \times \mathscr{D}$. For $b \in \{tt, ff\}$, the semantics of $[\![\psi]\!](\sigma, \omega) = (b, \sigma', \omega')$ iff $\langle \psi, \sigma, \omega\rangle \to \langle b, \sigma', \omega'\rangle$ likewise $[\![\varphi]\!](\sigma, \omega) = (b, \sigma', \omega')$ iff $\langle \varphi, \sigma, \omega\rangle \to \langle b, \sigma', \omega'\rangle$ using the following semantic rules.

[true] $$\frac{}{\langle true, \sigma, \omega\rangle \to \langle true, \sigma, \omega\rangle}$$

[equal] $$\frac{\langle expr_1, \sigma, \omega\rangle \to \langle val_1, \sigma, \omega''\rangle \quad \langle expr_2, \sigma, \omega''\rangle \to \langle val_2, \sigma, \omega'\rangle}{\langle expr_1 == expr_2, \sigma, \omega\rangle \to \langle b, \sigma, \omega'\rangle}$$ where $b = \begin{cases} tt & \text{if } val_1 = val_2 \\ ff & \text{otherwise} \end{cases}$

[not-equal] $$\frac{\langle expr_1, \sigma, \omega\rangle \to \langle val_1, \sigma, \omega''\rangle \quad \langle expr_2, \sigma, \omega''\rangle \to \langle val_2, \sigma, \omega'\rangle}{\langle expr_1 != expr_2, \sigma, \omega\rangle \to \langle b, \sigma, \omega'\rangle}$$ where $b = \begin{cases} tt & \text{if } val_1 = val_2 \\ ff & \text{otherwise} \end{cases}$

$$\text{[smaller]}\quad \frac{\langle expr_1, \sigma, \omega\rangle \to \langle val_1, \sigma, \omega''\rangle \quad \langle expr_2, \sigma, \omega''\rangle \to \langle val_2, \sigma, \omega'\rangle}{\langle expr_1 < expr_2, \sigma, \omega\rangle \to \langle b, \sigma, \omega'\rangle}$$

$$\text{where } b = \begin{cases} tt & \text{if } val_1 < val_2 \\ ff & \text{otherwise} \end{cases}$$

$$\text{[larger]}\quad \frac{\langle expr_1, \sigma, \omega\rangle \to \langle val_1, \sigma, \omega''\rangle \quad \langle expr_2, \sigma, \omega''\rangle \to \langle val_2, \sigma, \omega'\rangle}{\langle expr_1 > expr_2, \sigma, \omega\rangle \to \langle b, \sigma, \omega'\rangle}$$

$$\text{where } b = \begin{cases} tt & \text{if } val_1 > val_2 \\ ff & \text{otherwise} \end{cases}$$

$$\text{[larger-eq]}\quad \frac{\langle expr_1, \sigma, \omega\rangle \to \langle val_1, \sigma, \omega''\rangle \quad \langle expr_2, \sigma, \omega''\rangle \to \langle val_2, \sigma, \omega'\rangle}{\langle expr_1 >= expr_2, \sigma, \omega\rangle \to \langle b, \sigma, \omega'\rangle}$$

$$\text{where } b = \begin{cases} tt & \text{if } val_1 \geq val_2 \\ ff & \text{otherwise} \end{cases}$$

$$\text{[smaller-eq]}\quad \frac{\langle expr_1, \sigma, \omega\rangle \to \langle val_1, \sigma, \omega''\rangle \quad \langle expr_2, \sigma, \omega''\rangle \to \langle val_2, \sigma, \omega'\rangle}{\langle expr_1 <= expr_2, \sigma, \omega\rangle \to \langle b, \sigma, \omega'\rangle}$$

$$\text{where } b = \begin{cases} tt & \text{if } val_1 \leq val_2 \\ ff & \text{otherwise} \end{cases}$$

$$\text{[and]}\quad \frac{\langle \varphi_1, \sigma, \omega\rangle \to \langle b_1, \sigma, \omega''\rangle \quad \langle \varphi_2, \sigma, \omega''\rangle \to \langle b_2, \sigma, \omega'\rangle}{\langle \varphi_1 \;\&\&\; \varphi_2, \sigma, \omega\rangle \to \langle b, \sigma, \omega'\rangle}$$

$$\text{where } b = \begin{cases} tt & \text{if } b_1 = tt \text{ and } b_2 = tt \\ ff & \text{otherwise} \end{cases}$$

$$\text{[or]}\quad \frac{\langle \varphi_1, \sigma, \omega\rangle \to \langle b_1, \sigma, \omega''\rangle \quad \langle \varphi_2, \sigma, \omega''\rangle \to \langle b_2, \sigma, \omega'\rangle}{\langle \varphi_1 \;||\; \varphi_2, \sigma, \omega\rangle \to \langle b, \sigma, \omega'\rangle}$$

$$\text{where } b = \begin{cases} tt & \text{if } b_1 = tt \text{ or } b_2 = tt \\ ff & \text{otherwise} \end{cases}$$

$$\text{[not]}\quad \frac{\langle expr, \sigma, \omega\rangle \to \langle b', \sigma, \omega'\rangle}{\langle !\varphi, \sigma, \omega\rangle \to \langle b, \sigma, \omega'\rangle}$$

$$\text{where } b = \neg b'$$

$$\text{[clock larger-eq]}\quad \frac{\langle u, \sigma, \omega\rangle \to \langle val_r, \sigma, \omega\rangle}{\langle u >= c, \sigma, \omega\rangle \to \langle b, \sigma, \omega'\rangle}$$

$$\text{where } b = \begin{cases} tt & \text{if } val_r \geq c \text{ for } c \in \mathbb{N} \\ ff & \text{otherwise} \end{cases}$$

As the remaining rules for clock constraints are defined as expected and therefore are trivial, we leave out their formal definitions.

Note that the syntactic production `bexp` does not allow discrete boolean expressions to contain clock variables, nor does it allow the possible side effects (via function calls) to update the clock valuation. Thus the mapping imposed by $[\![\varphi]\!]$ is effectively $\mathscr{D} \to \{tt, ff\} \times \mathscr{D}$. Moreover, clock constraints are in-fact syntactically restricted to be side-effect-free, and can only refer clocks, which implies that the actual mapping is $\mathscr{C} \to \{tt, ff\}$.

Single Statements, Clock Assignments and Variable Assignments:

Given a statement λ from `single_stmt`, `single_act` or `single_asg`, its semantics $[\![\lambda]\!]$ is a mapping $\mathscr{C} \times \mathscr{D} \rightarrow \mathscr{C} \times \mathscr{D}$. The valuation of $[\![\lambda]\!](\sigma, \omega) = (\sigma', \omega')$ iff $\langle \lambda, \sigma, \omega \rangle \rightarrow \langle \sigma', \omega' \rangle$ wrt the following operational semantics:

[if-true] $$\frac{\langle \varphi, \sigma, \omega \rangle \rightarrow \langle b, \sigma, \omega'' \rangle \quad \langle stmt\_seq, \sigma, \omega'' \rangle \rightarrow \langle \sigma, \omega' \rangle}{\langle if(\varphi)\{stmt\_seq\}, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega' \rangle}$$

where $b = tt$

[if-false] $$\frac{\langle \varphi, \sigma, \omega \rangle \rightarrow \langle b, \sigma, \omega' \rangle}{\langle if(\varphi)\{stmt\_seq\}, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega' \rangle}$$

where $b = ff$

[while-true] $$\frac{\langle \varphi, \sigma, \omega \rangle \rightarrow \langle b, \sigma, \omega''' \rangle \quad \langle stmt\_seq, \sigma, \omega''' \rangle \rightarrow \langle \sigma, \omega'' \rangle \quad \langle while(\varphi)\{stmt\_seq\}, \sigma, \omega'' \rangle \rightarrow \langle \sigma, \omega' \rangle}{\langle while(\varphi)\{stmt\_seq\}, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega' \rangle}$$

where $b = tt$

[while-false] $$\frac{\langle \varphi, \sigma, \omega \rangle \rightarrow \langle b, \sigma, \omega' \rangle}{\langle while(\varphi)\{stmt\_seq\}, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega' \rangle}$$

where $b = ff$

[skip] $$\frac{}{\langle skip, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega \rangle}$$

[var-assign] $$\frac{\langle expr, \sigma, \omega \rangle \rightarrow \langle val, \sigma, \omega' \rangle}{\langle v = expr, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega'[v \mapsto val] \rangle}$$

[clock-assign] $$\frac{\langle expr, \sigma, \omega \rangle \rightarrow \langle val, \sigma, \omega' \rangle}{\langle u = expr, \sigma, \omega \rangle \rightarrow \langle \sigma[u \mapsto val], \omega' \rangle}$$

Sequences:

The semantics of sequences is defined by the following three rules. Given a variable action sequence α or a clock update sequence β, its semantics $[\![\alpha]\!]$ or $[\![\beta]\!]$ is a mapping $\mathscr{C} \times \mathscr{D} \rightarrow \mathscr{C} \times \mathscr{D}$. The semantics of α is that $[\![\alpha]\!](\sigma, \omega) = (\sigma', \omega')$ iff $\langle \alpha, \sigma, \omega \rangle \rightarrow \langle \sigma', \omega' \rangle$ wrt. the following operational rule:

[act-sequence] $$\frac{\langle \lambda, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega'' \rangle \quad \langle \alpha', \sigma, \omega'' \rangle \rightarrow \langle \sigma, \omega' \rangle}{\langle \alpha, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega' \rangle}$$

where $\alpha = \lambda\ \alpha'$ and λ must be expressed over *single_act*

The semantics of β is that $[\![\beta]\!](\sigma, \omega) = (\sigma', \omega')$ iff $\langle \beta, \sigma, \omega \rangle \rightarrow \langle \sigma', \omega' \rangle$ wrt. the following operational rule:

[act-sequence] $$\frac{\langle \lambda, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega'' \rangle \quad \langle \beta', \sigma, \omega'' \rangle \rightarrow \langle \sigma, \omega' \rangle}{\langle \beta, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega' \rangle}$$

where $\beta = \lambda\ \beta'$ and λ must be expressed over *single_asg*

Finally, the semantics of a statement sequence expressed over *stmt_seq* is defined as:

$$\text{[stmt-sequence]} \quad \frac{\langle \lambda, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega'' \rangle \quad \langle stmt\_seq', \sigma, \omega'' \rangle \rightarrow \langle \sigma, \omega' \rangle}{\langle stmt\_seq, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega' \rangle}$$

where $stmt\_seq = \lambda\ stmt\_seq'$

Function Calls:

In order to describe the semantics of calls to a function in the set of functions F, we introduce some extra notation. Since we do not use a notion of function declarations, we use $\mathcal{V}_f$ to denote the set of variables in the formal parameters of function f and $stmt_f$ to denote the body of function f, declared over the production `stmt_seq`. Since formal parameters of a function are local to that function, each function must be evaluated in a specific variable valuation ω^f, which is a mapping $\omega^f : \mathcal{V} \cup \mathcal{V}_f \rightarrow \mathbb{Z}$. To model return statements, a special variable *retval* is used to hold the value of the expression returned. Since *retval* is not in $\mathcal{V}$, *retval* can only be read and assigned through function calls.

$$\text{[funCall]} \quad \frac{\langle expr_1, \sigma, \omega \rangle \rightarrow \langle val_1, \sigma, \omega^1 \rangle \ldots \langle expr_n, \sigma, \omega^{n-1} \rangle \rightarrow \langle val_n, \sigma, \omega^n \rangle\ \langle stmt_f, \sigma, \omega^f \rangle \rightarrow \langle \sigma, \omega^{f'} \rangle}{\langle f(expr_1, \ldots, expr_n), \sigma, \omega \rangle \rightarrow \langle val, \sigma, \omega' \rangle}$$

where

Where $(\sigma, \omega^f) = (\sigma, \omega^n[v_i \mapsto val_i])$ for all $v_i \in \mathcal{V}_f$ and $\forall v \in \mathcal{V} : \omega' = \omega^{f'}$. Finally, $val = \omega'(retval)$ which is updated to hold the value of the return statement

$$\text{[return]} \quad \frac{\langle expr, \sigma, \omega \rangle \rightarrow \langle val, \sigma, \omega' \rangle}{\langle return\ expr, \sigma, \omega \rangle \rightarrow \langle \sigma, \omega'[retval \mapsto val] \rangle}$$

Notice that we require only a single *retval* variable since we do not allow recursion.

2.3.3 Extended Timed Automata

Having introduced the imperative syntax (and its semantics) used in this 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 composition $\alpha\beta$ of variable action sequences and clock assignment sequences. Where α and β are expressed over the syntactic productions `act_seq` and `asg_seq` respectively.

Guards and Invariants: The previously defined notion of guards and invariants has been extended 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 $\mathscr{C} \times \mathscr{D} \rightarrow \{tt, ff\}$.

Communication: Communication between automata is achieved by using a *channel* (like in CCS). Channels may be used to synchronize two or more automata. *Binary Synchronization*, as described for basic timed automata in Section 2.2, is the traditional behavior, in which case two automata synchronize on a channel.

In the extended formalism a notion of *Broadcast channels* are added, to enable synchronization between multiple timed automata. Broadcast channels differ from multi-synchronization, found in alternative literature, by the fact that broadcasts are non-blocking. Any number of receivers ($a?$) may choose to synchronize, but the sender ($a!$) is never blocked.

Urgency: In order to model the fact that time must not delay, we can choose to "mark" a channel or a location *urgent*. Informally, the effect of urgent channels is that whenever a location is reached, which has an outgoing edge using an urgent channel, synchronization must occur as soon as the edge is enabled. A location marked as urgent must not allow delays of any kind. Even more restricted behavior is achieved by marking a location *committed*. When entering a committed location, the next step in the transition system (defined formally below) must involve some committed location. That is, the behavior of the system is committed to the location.

In the following definition we use $\eta(\Phi(\mathcal{V}, F), \Psi(\mathcal{C}))$ to denote the set of all conjunctions over $\Phi(\mathcal{V}, F)$ and $\Psi(\mathcal{C})$.

Definition 4. *(The Extended Timed Automata)*
Let $\mathcal{A} = \langle L, l_0, \mathcal{V}, \mathcal{C}, \Sigma, F, E, I \rangle$ be a timed automaton extended with discrete variables.

- L is a finite set of locations, ranged over by l
- l_0 is the initial location
- $\mathcal{V}$ is a finite set of discrete variables, ranged over by v
- $\mathcal{C}$ is the set of clocks, ranged over by u
- Σ is the set of channels, ranged over by a
- F is a set of function declarations expressed in the above syntax.
- $E \subseteq L \times \eta(\Phi(\mathcal{V}, F), \Psi(\mathcal{C})) \times \Sigma \times Act(\mathcal{V}, F) \times Asg(\mathcal{C}) \times L$ is the set of edges
- $I : L \rightarrow \eta(\Phi(\mathcal{V}, F), \Psi(\mathcal{C}))$ assigns each location an invariant

Note: In order to refer to components of an edge e, we write $src(e)$, $guard(e)$, $channel(e)$, $update(e)$, $target(e)$ for l, g, a, r, l' respectively. Moreover, we use $in(l)$ to obtain the set of in-going edges for location l and dually, $out(l)$ obtains all outgoing edges for location l.

2.3.4 Semantics of the Extended Timed Automata

The semantics of the extended timed automata formalism is defined as a TLTS in Defintion 5. We use the notation $I^{\mathcal{C}}(l)$ to obtain ψ and $I^{\mathcal{V}}(l)$ to obtain φ in the invariant $I(l)$ for location l. For networks of timed automata we use $I^{\mathcal{V}}(\bar{l})$ to denote the conjunction of invariants: $I^{\mathcal{V}}(\bar{l}) = I^{\mathcal{V}}(l_1) \wedge \ldots \wedge I^{\mathcal{V}}(l)$ dually $I^{\mathcal{C}}(\bar{l}) = I^{\mathcal{C}}(l_1) \wedge \ldots \wedge I^{\mathcal{C}}(l)$.

Definition 5. *(Network of Extended Timed Automata)*
Let $\mathcal{A}_i = \langle L_i, l_i^0, \mathcal{V}_i, \mathcal{C}_i, \Sigma, F_i, E_i, I_i \rangle$ where $1 \leq i \leq n$ be a set of timed automata. A network of n timed automata $\mathcal{A}$ written $(\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n)$ is defined in terms of a transition system $\langle S, s_0, \rightarrow \rangle$ where $S = (L_1, \ldots, L_n) \times \mathscr{C} \times \mathscr{D}$ is the set of states and $s_0 = (\bar{l_0}, \sigma_0, \omega_0)$ is the initial state where $\bar{l}_0$ is the location vector $(l_1^0, \ldots, l_n^0)$.

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

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

- $\forall d'$ where $0 \leq d' \leq d : [\![I^{\mathcal{C}}(\bar{l})]\!](\sigma + d', \omega) = tt$.
- And $\forall\ d', l_i :\ d'$ does not enable an edge e for any l_i which is either *urgent* or *committed*. Furthermore, l_i does not have an outgoing edge with a channel which is both *urgent* and *broadcast*.

(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
- $[\![\psi]\!](\sigma, \omega) = tt$ and $[\![\varphi]\!](\omega) = tt$ where $\psi \wedge \varphi = g$ and
- $\omega' = [\![\alpha]\!](\omega)$ and $\sigma' = [\![\beta]\!](\sigma, \omega')$ where $\alpha\beta = r$
- $[\![I^{\mathcal{C}}(\bar{l}[l'_i/l_i])]\!](\sigma', \omega') = tt$ and $[\![I^{\mathcal{V}}(\bar{l}[l'_i/l_i])]\!](\omega') = tt$

(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 exists edges $l_i \xrightarrow{g_i,a!,r_i} l'_i$ and $l_j \xrightarrow{g_j,a?,r_j} l'_j$ such that
- $[\![\psi_i \wedge \psi_j]\!](\sigma, \omega) = tt$ and $[\![\varphi_j \wedge \varphi_j]\!](\omega) = tt$
- (output) $\omega'' = [\![\alpha_i]\!](\omega)$ and $\sigma'' = [\![\beta_i]\!](\sigma, \omega'')$ (followed by)
- (input) $\omega' = [\![\alpha_j]\!](\omega'')$ and $\sigma' = [\![\beta_j]\!](\sigma'', \omega')$.
- $[\![I^{\mathcal{C}}(\bar{l}[l'_i/l_i, l'_j/l_j])]\!](\sigma', \omega') = tt$ and $[\![I^{\mathcal{V}}(\bar{l}[l'_i/l_i, l'_j/l_j])]\!](\omega') = tt$

(bsync:) $\langle \bar{l}, \sigma, \omega \rangle \xrightarrow{a} \langle \bar{l}[l'_i/l_i, l'_j/l_j], \sigma', \omega' \rangle$ if:

- There exists an output edge $l_i \xrightarrow{g_i,a!,r_i} l'_i$ and
- for all l_j where $j \neq i$ we assume l_j to be input enabled, written $l_j \xrightarrow{g_j,a?,r_j}_* l'_j$ such that for l_j:
  - either $l_j \xrightarrow{g,a?,r} l'_j$ exists and is enabled or
  - l_j must take an implicate edge where $g = \neg(g_1 \vee \ldots \vee g_k)$ where $\{g_1, \ldots, g_k\} = \{g \mid l_j \xrightarrow{g,c?,r} l'_j\}$ and $r = \epsilon\epsilon$ and $l'_j = l_j$.
- $[\![\psi_i \wedge (\bigwedge_{j \neq i} \psi_j)]\!](\sigma, \omega) = tt$ and $[\![\varphi_i \wedge (\bigwedge_{j \neq i} \varphi_j)]\!](\omega) = tt$
- Let $\{1, \ldots, n\} \setminus \{i\} = \{j_1, j_2, \ldots, j_{n-1}\}$ in:
  - $\sigma' = \sigma_{n-1}$ for $\sigma_k = [\![\beta_{j_k}]\!](\sigma_{k-1}, \omega)$ where $1 \leq k \leq n-1$ and $\sigma_0 = [\![\beta_i]\!](\sigma, \omega)$
  - $\omega' = \omega_{n-1}$ for $\omega_k = [\![\alpha_{j_k}]\!](\omega_{k-1})$ where $1 \leq k \leq n-1$ and $\sigma_0 = [\![\alpha_i]\!](\omega)$
- $[\![I^{\mathcal{C}}(\bar{l}[l'_i/l_i, l'_j/l_j])]\!](\sigma', \omega') = tt$ and $[\![I^{\mathcal{V}}(\bar{l}[l'_i/l_i, l'_j/l_j])]\!](\omega') = tt$

The notion of a location l being input enabled on a channel a written $l \xrightarrow{g,a?,r}_* l'$, entails the assumption that the location have an implicit edge $l \xrightarrow{g\ a?\ r} l'$ such that $g = \neg(g_1 \vee \ldots \vee g_k)$ where $\{g_1, \ldots, g_k\} = \bigcup_{e \in out(l)} guard(e)$, $r = \epsilon\epsilon$ and $l = l'$. This eases the definition of broadcasts since we may now say that, whenever an automaton in the network outputs on a broadcast channel, all others synchronize using either an existing edge or the implicit edge, which is enabled exactly whenever all others are not. Using input enabled locations gives exactly the behavior where output on a broadcast channel is non-blocking and all able automata participate in the synchronization using their "real" edge whereas others use the implicit edge yielding no effect.

Model

The following defines a *model* for a network of extended timed automata, which we shall use in Chapter 4 to reason about correctness of our slicing approach.

Definition 6. *(Model)*
Let $\mathcal{S} = (S, s_0, \Sigma, \rightarrow)$ be the labeled transition system for a network of timed automata $\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$. The set AP, is a set of *atomic propositions* on the form: $\mathcal{A}_i.l$ where $l \in L_i$ or $\texttt{expr}_1 \sim \texttt{expr}_2 \in Expr(\mathcal{V}, F)$ where $\sim$ is a relational operator. A model of a network of extended timed automata, is a pair $\mathcal{M} = (\mathcal{S}, \mathcal{I})$ where $\mathcal{I}$ is a mapping function $\mathcal{I} : S \rightarrow 2^{AP}$. For $s = (l_1, \ldots l_n, \sigma, \omega) \in S$, $\mathcal{I}(s)$ is defined as follows: $\texttt{expr}_1 \sim \texttt{expr}_2 \in \mathcal{I}(s)$ iff $\omega \models \texttt{expr}_1 \sim \texttt{expr}_2$ and $\mathcal{A}_i.l \in \mathcal{I}(s)$ iff $l_i = l$.

Chapter

3

Model-Checking TA

This Chapter introduces the model-checking problem and various issues are highlighted. We give an introduction to the theory of region graphs and zones and how these can be represented using Difference Bound Matrices (DBMs). Finally, we look at the complexity of the model-checking problem.

Contents

3.1 Introduction

In the following Sections, the concepts of model-checking timed systems are introduced and various terms and techniques are described. Firstly, the logical query language CTL is described and compared to the approach used in UPPAAL. The UPPAAL engine implements a small subset of the TCTL language (but in practice suffiently large) because of the fact that efficiency is of paramount importance in UPPAAL. Furthermore, a general discussion on reachability analysis is presented, followed by a more thorough introduction to the concept of model-checking. Regions and zones are discussed intuitively in Section 3.3 and the underlying data-structure (the DBM) is discussed in Section 3.3.1. Finally, the complexity of model-checking is briefly introduced.

3.2 CTL

Computational Tree Logic (CTL) was introduced by Emerson and Clarke [23] as a specification language for finite-state systems. Let AP be a set of atomic propositions (see Definition 6) and N be the set of constants $\{0, 1, 2, \dots\}$ denoting the natural numbers. The formulae of CTL and TCTL are inductively defined as follows:

CTL: $\varphi := p \mid \mathbf{false} \mid \varphi_1 \rightarrow \varphi_2 \mid \exists X\varphi_1 \mid \exists(\varphi_1 \,\mathcal{U}\, \varphi_2) \mid \forall(\varphi_1 \,\mathcal{U}\, \varphi_2)$

TCTL: $\psi := p \mid \mathbf{false} \mid \psi_1 \rightarrow \psi_2 \mid \exists X\psi_1 \mid \exists(\psi_1 \,\mathcal{U}_{\sim c}\, \psi_2) \mid \forall(\psi_1 \,\mathcal{U}_{\sim c}\, \psi_2)$

where $p \in AP$, $\varphi_1, \varphi 2$ are CTL formulae, ψ_1, ψ_2 are TCTL formulae, $\sim \in \{<, \leq, =, \geq, >\}$ and $c \in N$.

$\exists X\varphi_1$ intuitively means that there is an immediate successor state, reachable by executing one step, in which φ_1 holds. $\exists(\varphi_1 \mathcal{U} \varphi_2)$ means that for some computation path, there exists an initial prefix of the path such that φ_2 holds at the last state of the prefix and φ_1 holds at all the intermediate states. $\forall(\varphi_1 \mathcal{U} \varphi_2)$ means that for every computation path the above property holds.

The model-checking engine of UPPAAL is designed to check a subset of TCTL formulae [1]. The abbreviations of UPPAAL from (T)CTL are as follows: `E<>`φ for $\exists(\mathbf{true}\ \mathcal{U}\ \varphi)$, `A[]`$\varphi$ for $\forall(\mathbf{true}\ \mathcal{U}\ \varphi)$, `E[]`$\varphi$ for $\neg$`A<>` $\neg\varphi$, `A[]`φ for $\neg$`E<>` $\neg\varphi$, and φ_1 `-->` φ_2 for `A[]` (φ_1 `imply A<>` φ_2). Furthermore, the query language of UPPAAL does not allow nesting of modalities, so only boolean combinations of atomic propositions or clock constraints are allowed for the subformulae ψ_1 and ψ_2.

3.2.1 Reachability Analysis

One of the most useful questions to ask about a timed automaton is the reachability of a given final state or a set of final states (corresponding to $E\Diamond\varphi$ properties in TCTL). The reachability problem is decidable [12]. In the verification of timed systems, a symbolic approach is adopted. The idea resembles symbolic model-checking for untimed systems, which uses boolean formulae to represent sets of states and operations on formulae to represent sets of state transitions. It is proven that the infinite state-space of timed automata can be finitely partitioned by symbolic states

using clock constraints known as *zones* (see Section 3.3). Model-checking concerns two types of properties, namely *liveness* and *safety*. The essential algorithm of checking liveness properties is loop detection, which is computationally expensive [11]. The main effort on verification of timed systems has been put on safety properties that can be checked using reachability analysis by traversing the state-space of timed automata.

Reachability analysis consists of two basic steps, computing the state-space of an automaton under consideration, and searching for states that satisfy or contradict given properties.

3.3 Model-Checking

With timed automata and the logic (T)CTL in hand, we wish to obtain a model checking algorithm able to automatically decide whether some formula holds for a timed automaton (or a network of timed automata).

In this context, the obvious difficulty is that timed automata have an infinite number of configurations, because there exists infinitely many possible clock valuations. This infiniteness has two sources

1. the clock values are unbounded
2. the set of real numbers is dense, even when restricted to a bounded interval.

Thus, the algorithm originally suggested for CTL [18, 38] does not apply.

The idea to overcome this difficulty can be described intuitively: Starting from two configurations (l, σ) and (l, σ') where σ and σ' are very close (fx $\sigma(u) = 1.234766$ and $\sigma'(u) = 1.23500$), a timed automaton will behave in roughly the same way and the two resulting configurations will verify the same (T)CTL formulae (assuming a notion of closeness for configurations is properly defined as a function of the formulae of interest). Also σ and σ' can be considered "close" when they are both beyond the constants handled by the timed automaton.

The idea of "closeness" is formally defined by an equivalence relation. Given the type of clock constraints appearing in the transitions and the largest constant used in these constraints, this equivalence $\equiv$ (see definition 7) on the clock valuations is defined with the following property: For any timed automaton using these constraints, two configurations (l, σ) and (l, σ') with $\sigma \equiv \sigma'$ satisfy the same (T)CTL formulae. For timed automata[1] the number of equivalence classes, called *regions* (see definition 7), are finite [12].

Definition 7. (Region) Let A be a timed automaton. We say that two clock valuations σ and σ' are equivalent, written $\sigma \equiv \sigma'$, iff

1. for each $u \in \mathcal{C}$, we have that either both $\sigma(u)$ and $\sigma'(u)$ are greater than c_u or

[1] Contrary to hybrid systems

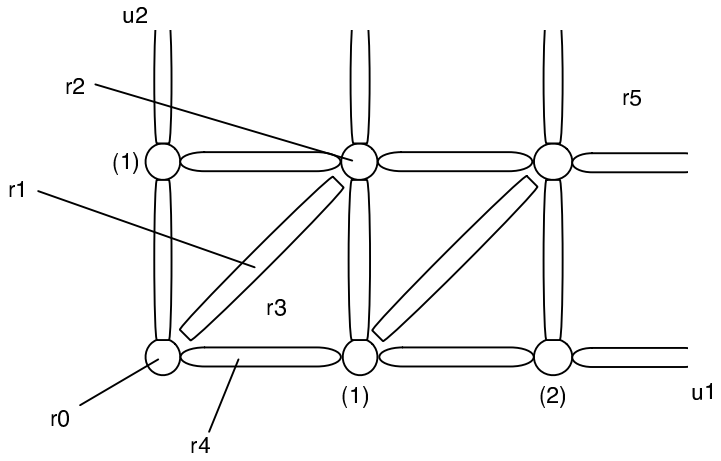


Figure 3.1: Regions for the constraints $u_1, u_2 \sim k$ with $k = 0, 1, 2$

$$\lfloor \sigma(u) \rfloor = \lfloor \sigma'(u) \rfloor$$

2. for each $u \in \mathcal{C}$ such that $\sigma(u) \leq c_u$ we have

$$frac(\sigma(u)) = 0 \text{ iff } frac(\sigma'(u)) = 0 \text{ and}$$

for all $u_1, u_2 \in \mathcal{C}$ such that $\sigma(u) \leq c_{u_1}$ and $\sigma(u_2) \leq c_{u_2}$ we have

$$frac(\sigma(u_1)) \leq frac(\sigma(u_2)) \text{ iff } frac(\sigma'(u_1)) \leq frac(\sigma'(u_2))$$

where $\lfloor \sigma(u) \rfloor$ denotes the integer part of the clock valuation and $frac(\sigma(u))$ denotes the fractional part of the clock valuation. Furthermore, c_u is defined as the largest constant against which the clock u is ever compared either in the guards or in the invariants present in A.

The equivalence relation $\equiv$ partitions the clock valuations of A into finitely many equivalence classes. Moreover, whenever σ and σ' are in the same equivalence class (that is, $\sigma \equiv \sigma'$ holds), then, for any location l in A, the configurations (l, σ) and (l, σ') are untimed bisimilar [41]. The equivalence classes induced by $\equiv$ are refered to as *regions*.

Example. Figure 3.1 represents the set of regions for two clocks where we are only interested in constraints of the form $u \sim k$ with $u \in \{u_1, u_2\}$ and $k = 0, 1, 2$ for u_1, $k = 0, 1$ for u_2. In this example, there are 28 regions. Two configurations belonging to the same region are "close".
Some regions amount to a single point, like `r0` (initial region), described by $u_1 = u_2 = 0$. Other regions are open surfaces in the plane, like region `r3`, described by $0 < u_2 < u_1 < 1$, or region `r5`, described by $u_1 > 2 \wedge u_2 > 1$. Lastly, the other regions are open half-lines or segments (like `r1` which corresponds to $0 < u_1 = u_2 < 1$).

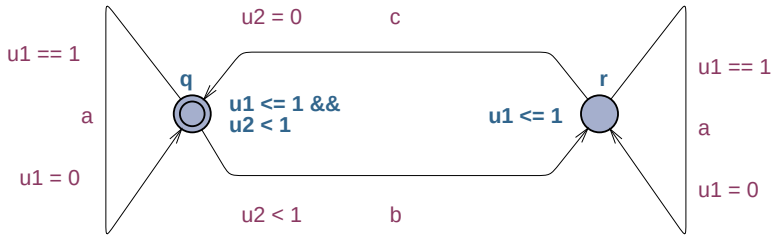


Figure 3.2: A timed automaton $\mathcal{A}$

The system starts in the initial region `r0`. As time passes, the system moves on to region `r1`, then on to the point $u_1 = u_2 = 1$ (region `r2`), etc., and up to region `r5`. If, rather than letting time elapse, we perform a discrete transition, the resetting of certain clocks leads to regions located on the axes. Fx, the reset of u_2 in region `r1` leads into `r4`, then `r3`, etc.

Rather than analyzing the infinite configuration graph, we analyze the finite graph of the "symbolic configurations" $(l, [\sigma])$ where $[\sigma]$ corresponds to the region of the valuation σ. This automaton, called *region graph*, is an abstract representation of the behavior of the timed automaton. We can use this type of model to determine the truth value of the (T)CTL formulae.

However, the problem with regions graphs is the potential explosion in the number of regions. In fact, it is exponential in the number of clocks as well as the maximal constants appearing in the guards of an automaton [11]. A more efficient representation of the state-space for timed automata is based on the notion of *zone* and *zone-graphs* [31, 22]. In a zone-graph, instead of regions, zones are used to denote symbolic states. This gives a coarser and more compact representation of the state-space. As an example, a timed automaton $\mathcal{A}$ and the corresponding zone-graph $G_{\mathcal{A}}$ (or reachability graph) is shown in figure 3.2 and 3.3, respectively. Note that the zone-graph has only 7 states. The region-graph for the same example would roughly comprise thirty states.

A zone is a clock constraint. Strictly speaking, a *zone* is the solution set of a clock constraint, that is the set of clock assignments satisfying the constraint. It is well-known that such sets can be efficiently represented an stored in memory as DBMs (Difference Bound Matrices)[10, 11] (See Section 3.3.1 on DBMs).

3.3.1 Difference Bound Matrix

A zone denoted by a clock constraint D is the set of clock assignments satisfying D. The most important property of zones is that they can be represented as matrices i.e. DBMs, which have a canonical representation (see Section 3.3.3).

To have a unified form for clock constraints, a reference clock $\mathbf{0}$ with the constant value 0 is introduced. Let $\mathcal{C}_0 = \mathcal{C} \cup \{\mathbf{0}\}$. Then any zone $D \in \mathcal{B}(\mathcal{C})$ (See definition in Section 2.1.2) can be rewritten as a conjunction of constraints of the form

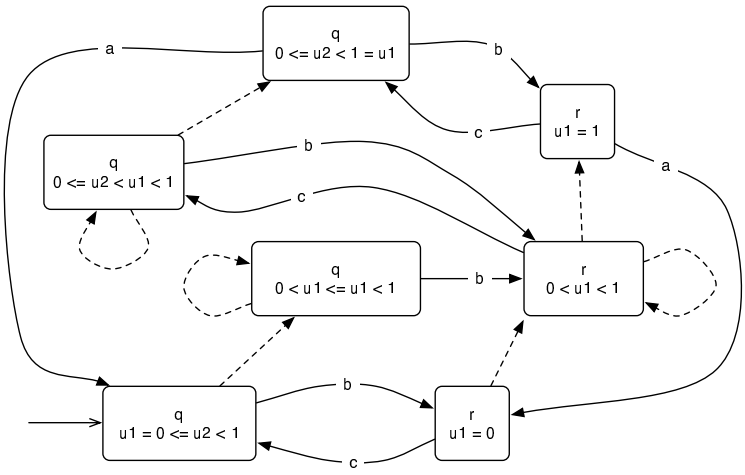


Figure 3.3: $G_{\mathcal{A}}$: A symbolic representation of $\mathcal{A}$

$u_1 - u_2 \prec n$ for $u_1, u_2 \in \mathcal{C}_0, \prec \in \{<, \leq\}$ and $n \in \mathbb{Z}$. Naturally, if the rewritten zone has two constraints on the same pair of variables only the strongest of the two is significant. Thus, a zone can be represented using at most $|\mathcal{C}_0|^2$ atomic constraints of the form $u_1 - u_2 \prec n$, such that each pair of clocks $u_1 - u_2$ is mentioned only once. Zones can then be stored using $|\mathcal{C}_0| \times |\mathcal{C}_0|$ matrices where each element in the matrix corresponds to a constraint. Since each element in such a matrix represents a bound on the difference between two clocks, they are named *Difference Bound Matrices* (DBMs). D_{ij} is used in the following to denote element (i, j) in the DBM representing the zone D.

To construct the DBM representation for a zone D, all clocks in $\mathcal{C}$ is numbered as $0, \ldots, n$ and the index for $\mathbf{0}$ is 0. Let each clock be denoted by one row in the matrix. The row is used for storing lower bounds on the difference between the clock and all other clocks, and thus the corresponding column is used for upper bounds. The elements in the matrix are then computed in three steps[11].

1. For each constraint $u_i - u_j \prec n$ of D, let $D_{ij} = (n, \prec)$.

2. For each clock difference $u_i - u_j$ that is unbounded in D, let $D_{ij} = \infty$. Where ∞ is a special value denoting that no bound is present.

3. Finally add the implicit constraints that all clocks are positive, i.e. $\mathbf{0} - u_i \leq 0$, and that the difference between a clock and itself is always 0, i.e. $u_i - u_i \leq 0$.

3.3.2 Maximum Bound Abstractions

The abstraction used in real-time model-checkers is based on the idea that the behaviour of an automaton is only sensitive to changes of a clock if its value is below a certain constant. That is, for each clock there is a maximum constant, such that once the value of a clock has passed this constant, its exact value is no longer relevant; only the fact that it is larger than the maximum constant matters.

Transforming a DBM to reflect this idea is often referred to as *extrapolation* [13, 8] or *normalisation* [21]. In Figure 3.2 the maximum constant for u_1 and u_2 is 1, since 1 is the maximal constant to which both u_1 and u_2 are compared.

3.3.3 Canonical DBMs

Usually there is an infinite number of DBMs sharing the same solution set. However, for each class of DBMs with the same solution set, there is a canonical one in which no atomic contraint can be strengthened without losing solutions.
To compute the canonical form of a given DBM, the tightest constraint on each clock difference needs to be derived. This problem can be solved using a weighted graph representation of the DBMs where the clocks are nodes and the constraints are edges labeled with bounds. A constraint on the form of $u_1 - u_2 \prec n$ will be converted to an edge from node u_1 to node u_2 labeled with $(n, \prec)$, namely the distance from u_1 to u_2 is bounded by n. The tightest constraint for a given pair of clocks in a DBM is equivalent to finding the shortest path between their nodes in the graph interpretation of a corresponding DBM [11]. Finding the canonical form of a difference bound matrix can be automated by using the Floyd-Warshall algorithm [48], which has cubic complexity. The algorithm guarantees that all the possible combination of indices are systematically checked to determine if further tightening is possible.
After the DBM has been converted to canonical form, it can be determined if the corresponding clock zone is nonempty by examining the entries on the main diagonal of the matrix. If the clock zone described by the matrix is nonempty, all of the entries along the main diagonal will have the form $(0, \leq)$. If the clock zone is empty or unsatisfiable, there will be at least one negative entry on the main diagonal.

3.3.4 Operations on DBMs

In order to do symbolic state-space exploration of timed automata, various operations are needed.
The basic operations on DBMs can be devided into two classes:

- **Property-Checking:** This class includes operations to check the consistency of a DBM, the inclusion between zones, and whether a zone satisfies a given constraint.

- **Transformation:** This class includes operations to compute the strongest postcondition and the weakest precondition of a zone according to conjunction with guards, time delay and clock reset.

After performing any of these operations, the resulting matrix may fail to be in canonical form. Thus, as a final step, the matrix must be reduced to canonical form again. The basic operations can be implemented efficiently (Essentially, it has the same complexity as transforming the DBM into canonical form, i.e. cubic in the number of clocks). Moreover, the implementation of these operations is relatively straightforward to program [11].

3.4 Complexity

Model-checking suffers from the so-called *state explosion* problem, which is, that the need of resources for verification grows exponentially with the model to verify (worst case). For timed systems, the problem is more serious. It is known that the model checking problem for timed systems is PSPACE-complete [1].[2] The number of regions grows exponentially with the number of clocks: For n clocks and for constraints in which every constant k is upperbounded by M, the number of regions is $O(n!M^n)$ [1, 2]. Though there are even more zones, in practise, this has proven a far more efficient data structure, as the exact collection of zones generated is heavily guided by the actual timed automata analysed [7].

[2] Intuitively, a problem is PSPACE-complete if it is one of the hardest among all the problems solvable using polynomial memory space. In practice, any known algorithm will use up exponential time in the worst case. (By comparison with the more commonly known NP-completeness notion, PSPACE-complete problems do not even possess an efficient test that a purported solution discovered by chance or otherwise is indeed a solution [45].

Chapter

4 Slicing

In this Chapter, we introduce slicing for the purpose of syntactic reduction of the extended time automata formalism given in Chapter 2. The developed slicing theory will allow us to implement slicing for the Uppaal tool from which we present performance results in later chapters.

Contents

4.1 Introduction

The original concept of slicing was introduced by Weiser in [52]. According to Weiser, *a slice corresponds to the mental abstractions that people make when they are debugging a program.* Various notions of program slices have been proposed, most of which are compared by Tip in [51] along with numerous methods and supporting theory to compute program slices. Weiser defined a program slice S as a reduced, executable program obtained from a program P by removing statements, such that S replicates part of the behavior of P i.e. the program slice S is a fragment of the original program with regards to some criteria, called the slicing criteria. Another common definition from [51] is that a slice, is a subset of the statements and control predicates of the program that directly or indirectly affect the values computed at a certain point in the program, but that do not necessarily constitute an executable program. In Weisers original approach, slices are created by computing consecutive sets of transitively relevant statements, according to *data-* and *control-flow* dependencies.

4.1.1 Slicing Timed Automata

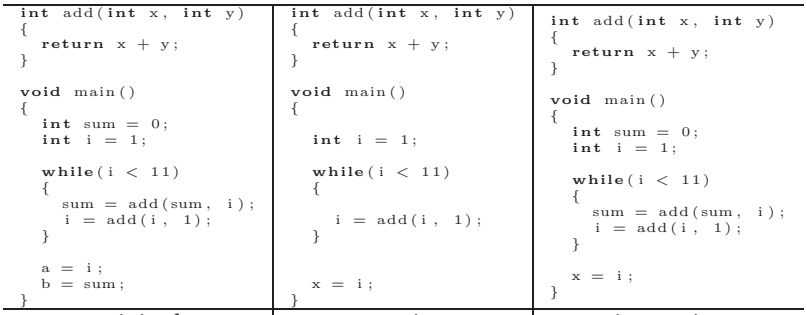
Traditionally, slicing has been suggested for many applications [51] primarily related to *testing* or *debugging.* We here show how to use slicing for syntactic reduction of models expressed, using the extended definition of Timed Automata given in Section 2.3.

In addition to the imperative control structures, the main difficulty of models in the extended Timed Automata language is handling dependencies across function calls. This problem is known in existing slicing theory [35] as *the calling context problem.* Intuitively, the problem describes the fact that a function with multiple call-sites may contribute to the production of unnecessary large slices when using conservative algorithms, like the one first presented by Weiser in [52]. The static analysis employed in such algorithms may conclude that the control-flow (see Section 4.2.2) may enter the procedure from one call-site and return at another - which is not in accordance with the natural control-flow of programs. To overcome this problem, the algorithm presented here is based on work from [24, 35, 46, 47] which employ the use of *System Dependency Graphs* (SDG) and summary information. The following Section illustrates this problem by example. In Section 4.2.3, we present more thoroughly the notion of an SDG and the use of *summary edges* to achieve precise slices.

Example of The Calling Context Problem

Figure 4.1 illustrates the results of slicing the application with respect to the statement ⟨`x = i;`⟩. Given the original program on the left, the desired slice with respect to ⟨`x = i;`⟩ is given as the middle instance (denoted precise). The instance on the right is a slice computed using the slicing algorithm presented by Weiser, where the algorithm has determined the relevance of function `add` and have mistakenly followed a path in the control-flow graph (see Section 4.2.2) which have entered function `add` at the statement `add(i,1)` and returned along a control-flow edge from `add` to `add(sum, i)` which has "fooled" the algorithm into adding the decla-

ration of `sum`. Such traces may be avoided by keeping track of the *enter-exit* relation ship of function calls also know as *the calling context problem*.



```
int add(int x, int y)
{
  return x + y;
}

void main()
{
  int sum = 0;
  int i = 1;

  while(i < 11)
  {
    sum = add(sum, i);
    i = add(i, 1);
  }

  a = i;
  b = sum;
}
```

original

```
int add(int x, int y)
{
  return x + y;
}

void main()
{

  int i = 1;

  while(i < 11)
  {

    i = add(i, 1);
  }


  x = i;
}
```

precise

```
int add(int x, int y)
{
  return x + y;
}

void main()
{
  int sum = 0;
  int i = 1;

  while(i < 11)
  {
    sum = add(sum, i);
    i = add(i, 1);
  }

  x = i;
}
```

imprecise

Figure 4.1: Example on slicing sequential code

4.1.2 Benefits of Slicing Timed Automata

The main idea is that slicing timed automata models is interesting primarily for the purpose of reducing the state-space explored by the model-checker, which consequently increases the performance of the verification process.

The slicing approach presented in this thesis is inspired by closely related work by Janowska and Janowski [37], who introduce slicing for timed automata with discrete data (simple updates of integer variables). But since the definition of timed automata presented here includes a complete imperative language used for manipulating discrete variables and clocks, a more sophisticated approach is required. We show in Section 4.2.3 that techniques used in slicing of traditional sequential code [46, 47] may be used for this purpose.

4.2 Preliminary Definitions

Before moving on to the central slicing algorithms and definitions, we introduce required terminology and definitions.

4.2.1 Statements and Variables

In the remainder of this thesis, we use the term statement, represented by λ, to denote an *occurrence* in a model of a single statement, clock assignment, variable assignment or function call, expressible in the `single_act`, `single_asg` or `single_stmt` productions on page 23.

Moreover, we use $\Lambda(\alpha)$ and $\Lambda(\beta)$ to denote the set of statements occurring in the sequences α and β respectively and $\Lambda(e) = \Lambda(\alpha) \cup \Lambda(\beta)$ where $\alpha\beta = \mathit{update}(e)$ to denote the set of statements occurring in the update of e. Similarly, we use $\Lambda(f)$ to denote the set of statements occurring in function f.

Definition 8. *(Statement Sets)*
We define the set of statements obtained by Λ by induction in the syntax such that:

- for λ in $Act(\mathcal{V}, F)$ or $Asg(\mathcal{C})$, $\Lambda(\lambda) = \{\lambda\}$ and
- for *return* statements, we have that $\Lambda(\texttt{return}\ \text{expr}) = \{\texttt{return}\ \text{expr}\}$
- for *if* and *while* statements, with empty bodies (ϵ), we have that:
  - $\Lambda(\texttt{if}(\varphi)\{\epsilon\}) = \{\texttt{if}(\varphi)\{\}\}$ and
  - $\Lambda(\texttt{while}(\varphi)\{\epsilon\}) = \{\texttt{while}(\varphi)\{\}\}$

  and in the case of non-empty bodies we have that
  - $\Lambda(\texttt{if}(\varphi)\{\lambda_1\lambda_2\ldots\lambda_n\}) = \Lambda(\lambda_1\lambda_2\ldots\lambda_n) \bigcup \Lambda(\texttt{if}(\varphi)\{\epsilon\})$ and
  - $\Lambda(\texttt{while}(\varphi)\{\lambda_1\lambda_2\ldots\lambda_n\}) = \Lambda(\lambda_1\lambda_2\ldots\lambda_n) \bigcup \Lambda(\texttt{while}(\varphi)\{\epsilon\})$
- for a sequence of statements $\lambda_1\lambda_2\ldots\lambda_n$ expressed from any production, we have that $\Lambda(\lambda_1\lambda_2\ldots\ \lambda_n) = \bigcup_{\lambda_i} \Lambda(\lambda_i)$ where $i \in \{1, \ldots, n\}$
- for a function with an empty body we say that:
  - $\Lambda(\texttt{foo(p,...,q)}\{\epsilon\}) = \emptyset$

  and with a non-empty body:
  - $\Lambda(\texttt{foo(p,...,q)}\{\lambda_1\lambda_2\ldots\lambda_n\}) = \Lambda(\lambda_1\lambda_2\ldots\lambda_n)$

Finally the set of all occurring statements in a timed automata $\mathcal{A} = \langle L, l_0, \mathcal{V}, \mathcal{C}, \Sigma, F, E, I\rangle$ is defined as:

$$\Lambda(\mathcal{A}) = \bigcup_{e\in E} \Lambda(e) \cup \bigcup_{f\in F} \Lambda(f).$$

Variables:

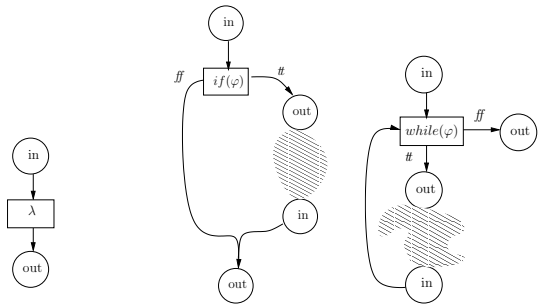
For the purpose of the analysis introduced in this Chapter, it will be sufficient to refer to variable names without referring to their type. Therefore, we will not discriminate between variables representing clocks or discrete data. For this purpose we will use V to denote the union of discrete variables from $\mathcal{V}$ and clock variables in $\mathcal{C}$. Also, we will use z to range over V.

4.2.2 Control-Flow

The extensions made to timed automata in this thesis (and in the complete UPPAAL language) creates an interesting hybrid of timed automata and traditional imperative code, in which the control flow of a model is influenced by traditional imperative branching, loop statements and by the structure of the automaton. The control-flow of the automata is expressed explicitly by edges and locations and thus requires no further explanation. In the following, we discuss the control-flow of the imperative language and the induced control-flow graph (CFG).

In the following figures, vertices (rectangular boxes) represent occurrences of statements and the circled (*in*) and (*out*) represents the *program points* where graph fragments may be joined to model the control-flow of imperative code. We say that each edge in the control-flow graph represents a program point p and use the notation, $\xrightarrow{p_{i-1}} \lambda_i \xrightarrow{p_i}$ to describe the fact that program point p_i follows immediately after statement λ_i where as p_{i-1} is immediately prior to λ_i.

The following gives the control-flow graphs known from classic static analysis [50]. Figure 4.2(a) illustrates the basic "building block" of the control-flow, which represents a single statement λ from $Act(\mathcal{V}, F)$ or $Asg(\mathcal{C})$. Figure 4.2(b) illustrates the structure of *if* statements which may skip a *sequence* of statements (illustrated by the "cloud") based on the valuation of the boolean expression φ. Likewise 4.2(c) shows how *while* statements may repeat a *sequence* of statements based on the valuation of the condition φ.



(a) single (b) if (c) while

Figure 4.2: Control-flow of simple imperative constructs

In order to define the control-flow of function calls, we exploit the fact that any expression, expressed in the imperative language considered here (see Section 2.3.1), may be rewritten to use temporary variables which are assigned by the original function call. This allows us to define the control-flow of functions in a generic manner (i.e. as a statement), ignoring the fact that function calls may be either expressions or statements.

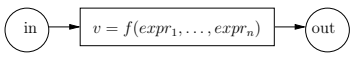


Figure 4.3: Function call statement

The control-flow imposed by function calls is defined in terms of in-lined functions. That is, the program points where the (in) and (out) of the function call statement is connected to the remaining CFG is replaced by (in) and (out) of the function.

Figure 4.4 shows the control-flow of in-lined functions replacing statements such as the one shown in Figure 4.3. Figure 4.4 illustrates how the actual parameters

of the call are assigned to the formal parameters of the function declaration, after which the body of the function is entered at (out'). Having executed the body, an arbitrary number of return statements in the body are replaced by assignments to the variable v after which the control-flow is returned to the original program point (out).

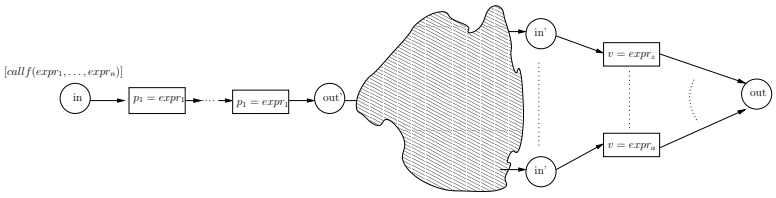


Figure 4.4: Control-flow of in-lined function call $v = f(expr_1, \ldots expr_n)$

Although we continue to make a clear distinction between the structure of the automata and the control-flow of the imperative constructs, Figure 4.5 illustrates how the control-flows of the automata and the imperative language may be viewed combined. Notice, that since the definition of updates $\alpha\beta$ on edges only allow sequences of statements, which are either assignments or function calls, loops and branches are only present in the control-flow within the body of a function, called in the update.

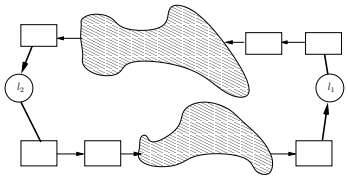


Figure 4.5: The combined control flow of a simple automata. The clouds illustrate how calls to functions may introduce a more complex control-flow structure.

Control-Flow Edges

An edge in a control-flow graph is written $\lambda \xrightarrow{cfg} \lambda'$ whenever it connects two statements.

Formally, a CFG is a graph where vertices are either statements or locations and there exists an edge in the CFG iff one of the following hold:

- Whenever λ is the first statement in the sequence α for an update $\alpha\beta = update(e)$ and $l = src(e)$, an edge connects $l \xrightarrow{cfg} \lambda$.
- Whenever λ is the last statement in the sequence β for an $\alpha\beta = update(e)$ and $l = target(e)$, an edge connects $\lambda \xrightarrow{cfg} l$.
- Whenever λ follows λ' in a sequence, an edge connects $\lambda' \xrightarrow{cfg} \lambda$.

- Whenever λ is the first statement evaluated after the condition φ of a control statement λ', an edge connects $\lambda' \xrightarrow{cfg} \lambda$
- Whenever a statement $\lambda' \equiv v = f(expr_1, \ldots, expr_n)$ follows statement λ in a sequence, an edge connects $\lambda \xrightarrow{cfg} \lambda_1$ where λ_1 is an assignment, assigning the first actual parameter of the call, to the first formal parameter of f.
- Whenever a statement $\lambda' \equiv v = f(expr_1, \ldots, expr_n)$ precedes statement λ in a sequence an edge connects $\lambda_h \xrightarrow{cfg} \lambda$ for each λ_h representing *return* statements, all of which are replaced by assignments $v = expr$, where $expr$ is the expression which is returned by the original *return* statement.
- Whenever a function f has formal parametes id_i, id_j for $i > 0$ and $j = i + 1$ an edge connects $\lambda_i \xrightarrow{cfg} \lambda_j$ where λ_i is an assignment to id_i and λ_j is an assignment to id_j with the actual parameters $expr_i$ and $expr_j$ at the call to f.

Paths

The term path is used ambiguously throughout the remainder of this Chapter. We shall refer to paths within the control-flow of a statement as described above; a path in the automata i.e. locations and edges and as a path in the combined control-flow as illustrated in Figure 4.5. Finally, we shall also refer to paths in an SDG which is presented in Section 4.2.3. Although this ambiguity exists, it should be clear from the context which type of path is referred to.

There exists a path in the CFG from statement occurrence λ to statement λ' written $\lambda \xrightarrow{cfg}_* \lambda'$, if there is a sequence of statements connected by CFG edges $\lambda_1 \xrightarrow{cfg} \lambda_2 \xrightarrow{cfg} \ldots \xrightarrow{cfg} \lambda_m$ in some control-flow graph where $\lambda = \lambda_1$ and $\lambda' = \lambda_m$.

We shall use π to denote a path in the automaton from a location $l_1 \in L_i$ to location $l_m \in L_i$ and $path(l_1, l_m)$ to compute the set of paths from l_1 to l_m. A path π is a sequence of locations and edges on the form $l_1 e_2 l_2 \ldots e_m l_m$ where $m \geq 2$ and for all l_j, e_j where $j \in \{2, \ldots, m\}$ it holds that $l_j \in L_i$ and $e_j \in out(l_{j-1}) \cap in(l_j)$.

Reaching

Reaching is a standard term in static analysis. It describes the fact that a defintion of (assignment to) a variable z at some statement occurrence λ has not been overridden when the program execution arrives at some other statement λ' which references (reads) z. That is, the value assigned to z is preserved over some control flow, at least until λ'. In Definition 9, we use the functions $ref(\lambda)$ and $def(\lambda)$ to denote the sets of variables referenced and defined by statement λ.

To define *def()* and *ref()* (see Table 4.1) we use the function $vars(expr)$ to obtain the set of variables occurring in $expr$. $vars(expr)$ is defined inductively as:

$$vars(m) = \emptyset \qquad vars(z) = \{z\}$$

$$vars(expr_1 \otimes expr_2) = vars(expr_1) \cup vars(expr_2)$$

$$vars(-expr) = vars(expr)$$

$$vars(f(expr_1, \ldots, expr_n)) = \bigcup_{i \in \{1,\ldots,n\}} vars(expr_i)$$

$vars(expr)$ is extended to $vars(\varphi)$ and $vars(\psi)$ in the intuitive way.

$\lambda \equiv if(\varphi)\{\ldots\}$	$ref(\lambda) = vars(\varphi)$	$def(\lambda) = \emptyset$
$\lambda \equiv while(\varphi)\{\ldots\}$	$ref(\lambda) = vars(\varphi)$	$def(\lambda) = \emptyset$
$\lambda \equiv return\ expr$	$ref(\lambda) = vars(expr)$	$def(\lambda) = \emptyset$
$\lambda \equiv y = expr$	$ref(\lambda) = vars(expr)$	$def(\lambda) = \{y\}$
$\lambda \equiv skip$	$ref(\lambda) = \emptyset$	$def(\lambda) = \emptyset$
$\lambda \equiv funcall(expr_1, \ldots, expr_n)$	$ref(\lambda) = \bigcup_{i \in 1\ldots n} vars(expr_i)$	$def(\lambda) = \emptyset$

Table 4.1: Functions $ref(\lambda)$ and $def(\lambda)$.

Since the control-flow of the formalism presented here is a hybrid of automata and sequential code, the standard definition of reaching must be extended slightly. The first case of the definition describes reaching within the traditional control-flow of imperative code. The second case describes a combination of the control-flow for automata and imperative code. The third case describes how statements may influence other statements across parallel automata. Figure 4.6 illustrates this.

Definition 9. *(Reaching)*
Let $\mathcal{A}$ be a network of extended timed automata $\mathcal{A}_i$ for $1 \leq i \leq m$ and $V = \bigcup_i \mathcal{V}_i \cup \mathcal{C}_i$ be the set of all variables. For $z \in V$ and $e_1 \in E_i, e_2 \in E_j$ where $i, j \in \{1, \ldots, m\}$ We say that the definition of z in an occuring statement $\lambda \in \Lambda(e_1)$ is *reaching* for $\lambda' \in \Lambda(e_2)$ iff $z \in def(\lambda) \cap ref(\lambda')$ and one of the following holds:

1. $e_1 = e_2$ and there exists a path $\lambda_1 \xrightarrow{cfg} \lambda_2 \xrightarrow{cfg} \ldots \xrightarrow{cfg} \lambda_m$ where $\lambda = \lambda_1$ and $\lambda' = \lambda_m$ and $\forall \lambda_j : z \notin def(\lambda_j)$ where $j \in \{2, \ldots, m-1\}$
2. There exists a path $\pi \in path(l, l')$ where $l = src(e_1)$ and $l' = target(e_2)$ such that:
   a) $\forall \lambda'' \in \Lambda(e_1)$ s.t. there exists a path $\lambda \xrightarrow{cfg} \lambda'' : z \notin def(\lambda'')$ and
   b) $\forall \lambda''' \in \Lambda(e_2)$ s.t. there exists a path $\lambda''' \xrightarrow{cfg} \lambda' : z \notin def(\lambda''')$.
   c) Finally, $\forall e_i \in \pi$ such that $e_i \neq e_1$ and $e_i \neq e_2$, $z \notin def(\lambda'''')$ for any $\lambda'''' \in \Lambda(e_i)$.
3. $i \neq j$ (i.e. reaching between parallel automata is very conservative)

In order to reflect the flow of information to the guards of the model, we extend the notion of reaching to include the fact that a statement λ, in the control-flow of an update, may also reach a guard. We need only to extend the $ref()$ and $def()$ functions in such a way that $ref(\varphi) = vars(\varphi)$ and $ref(\psi) = vars(\psi)$ and since guards are side-effect free we must have that $def(\varphi) = def(\psi) = \emptyset$. We may now

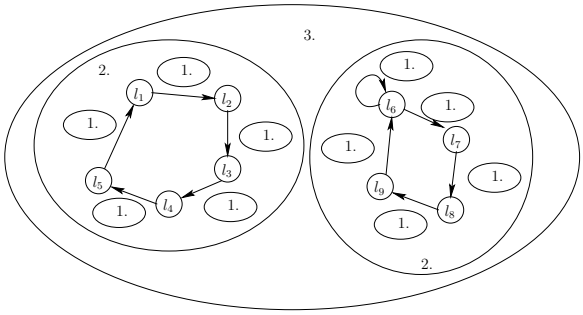


Figure 4.6: Illustrates the impact of the three levels of reaching.

add to the reaching definition a fourth case.

A statement $\lambda \in \Lambda(e_1)$ may reach $guard(e_2)$ if $\exists z \in V$ s.t. $z \in def(\lambda) \cap \big(ref(\varphi) \cup ref(\psi)\big)$ where $guard(e_2) = \varphi \wedge \psi$ and the following hold.

4. $\exists \pi \in path(l, l')$ where $l = src(e_1)$ and $l' = target(e_2)$ and $\forall \lambda' \in \Lambda(e_1)$ s.t. there exists a path $\lambda \xrightarrow{cfg} \lambda' : z \notin def(\lambda')$ and $\forall e_i \in \pi$ where $e_i \neq e_1 \wedge e_i \neq e_2$ it holds that $\forall \lambda'' \in \Lambda(e_i) : z \notin def(\lambda'')$

4.2.3 Dependencies

The slicing technique presented here is based on the concept of a *System Dependency Graph* [35] (SDG), which we use to represent the dependencies in a network of extended timed automata. Intuitively, slices are constructed from the set of vertices (statements) reachable in the SDG using some traversal strategy. As an example, Figure 4.7 shows the SDG for the code in Figure 4.1 (page 43). The following introduces the concept of a *function dependency graph*, which is the basic component of the SDG.

Function Dependency Graph

A *function dependency graph*, (FDG) is a graph representing the local dependencies within a function or updates on edges. It is a directed graph connected by two kind of edges (control- and flow edges) whose vertices represent statement occurrences. Moreover, an FDG contains a unique shadow[1] vertex called the *entry* vertex, which is added to represent the entrance of a function.

Edges in the FDG represent dependencies among occurring statements within a function or the updates of an automaton. An edge represents either *control* or *data-flow*. The FDG contains a *control* edge from vertex λ to vertex λ' iff one of the following holds:

- λ is the entry vertex and λ' is not subordinate to any control statement i.e. *while* or *if* statement.

[1] As in [50] we use the term shadow vertex to denote a vertex representing an artificial statement which is not part of the original program.

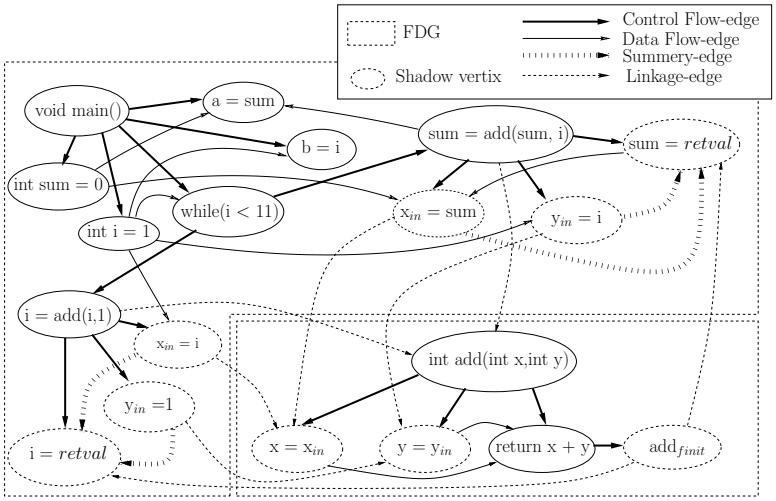


Figure 4.7: The system dependency graph of program main, containing Control-, Data- and Call-Dependencies - extended with summary edges (see page 52).

- λ represents a control statement and λ' represents a statement immediately subordinate to the control structure represented by λ.

The FDG contains a *data-flow* edge from λ to λ' iff the definition of some variable in λ reaches λ', as defined by case 1 in Definition 9.

A Note on Control Edges: It may be noted that control edges represent dominance of statements which is easily computed from the control-flow graphs as introduced in Section 4.2.2. Control edges form a tree (or hierarchy) of statements, such that, whenever a statement controls the execution of others, an edge is added from the controlling node to the controlled. It should be intuitive (from the control-flow graphs of Figure 4.2, how to produce such a hierarchy for *if* and *while* statements. The only vertices which are not dominated by others are function entry-points, which in-turn forms the root of the subordinate tree.

Special FDGs: In-order to adapt the FDG/SDG approach to slicing the extended timed automata formalism, a special case FDG is created for the automaton. As the behavior of a model is primarily defined by the structure of the automaton, it may be thought of as the "main" function (entry-point), which in turn calls the auxiliary functions at designated points in its execution. To model the dependencies between update statements, guards and invariants, a specialized FDG is constructed. Although the edges of the FDG remain the same, computing data-flow edges (reaching), becomes somewhat more complex, than in traditional imperative code. Although a formal description of this computation is not within the scope of this thesis, Section 5.3.4 describes, informally, how this may be implemented for Uppaal, by computing the fix-point of reaching statements for each program point of the automaton.

System Dependency Graph

The definition of the SDG introduced here may be distinguished from other definitions by the fact that the SDG, is based not only on the control-flow of imperative components, but also the structure of the timed automata. That is, our definition of the SDG models a language with the following properties:

- A system consists of a network of timed automata with a unique start location and for each edge associated with a timed automata, a sequence of statements is executed, possibly calling a set of common auxiliary functions.
- A set of global functions having pass-by-value parameters and return statements.
- A set of global variables which are either real or integer valued.

The SDG is a "super" graph of all the FDGs for a network of timed automata connected by additional edges to handle function calls. The approach used here, inspired by [35] is based on a notion of *function linkage*. In order to link FDGs, we add four new types of vertices to the FDG, called shadow vertices:

- An *initialization* vertex: For each formal parameter of a function, a vertex is added, representing the assignment of the actual parameter to the formal parameter.
- A *finalization* vertex: For each *return* statement, a vertex is added to represent the valuation of the expression to be returned.
- A *pre-processing* vertex: For each parameter given in a function call, a vertex is added to represent the valuation of the actual parameter.
- A *post-processing* vertex: For each assignment from a function call (i.e. whenever a variable is assigned the value returned by a function call), a vertex is added to represent the assignment at the return value.

Value passing in function calls is represented using the *pre-processing* and *post-processing* vertices. These vertices are connected subordinate to the function call vertex, which is referred to as the *call-site* vertex, using control edges. Likewise, the *initialization* vertices are connected using control edges subordinate to the *entry* vertex of the function. Finally, *finalization* vertices are connected using a control edge to the *return* statement.

Linkage Edges: Additional linkage edges are added such that:

- A directed *call-edge* starting at the *call-site* vertex connects the *call-site* vertex to the *entry* vertex.
- A directed *actual-in-edge* starting at the *pre-processing* vertex connects the *pre-processing* vertex to the corresponding *initialization* vertex.
- A directed *actual-out-edge* edge starting at the *finalization* vertex connects the *finalization* vertex to the corresponding *post-processing* vertex.

Definition 10. *(System Dependency Graph)*
The SDG is defined in terms of *function dependency graphs* (FDGs) and linkage edges. The SDG for a network of n timed automata $\mathcal{A}$, contains:

- An FDG for each function, $f \in \bigcup_{i \in \{1,\ldots,n\}} F_i$
- An FDG for each timed automata $\mathcal{A}_i : 1 \leq i \leq n$
- *Call-edges* from all *call-site* vertices to the respective *entry* vertex.
- *actual-in-edges* from all *pre-processing* vertices to the matching *initialization* vertex at the *entry* vertex of the called function.
- *actual-out-edges* from all *finalization* vertices to the matching *post-processing* vertex of the *call-site*.

Summary-Edges

As will be described in Section 4.4, it is possible to employ a more sophisticated (compared to the simple transitive closure) slicing algorithm to avoid the *calling-context-problem* and thereby obtain a more precise slice [35]. The algorithm which we shall introduce as Algorithm 2, avoids the calling context problem by ignoring certain linkage edge at different points in the traversal of the SDG. This approach would result in a too coarse reduction in the exploration without the use of summary edges. Although a formal definition is outside the scope of this thesis, the following presents the main idea.

For each actual parameter of a function call, the subordinate graph of its *pre-processing* vertex is analyzed using a notion of *same-level realizable paths* [47], which ensures that only realistic call paths are explored. The purpose of this analysis is to "short-circut" the transitive exploration which is computed by standard *worklist* algorithms, such as Algorithm 1, by statically computing how data-flow from the actual parameter may influence the return value of the call. Whenever the return value is dependent on the actual parameter, a *summary-edge* is added from the *pre-processing* vertex to the *post-processing* vertex. Using this (summary) information, the algorithm will, starting at the *post-processing* vertex, be able to explore the predecessor vertices of the *pre-processing* vertices without traversing the called function.

SDG Paths A path in an SDG is a sequence of statements and edges. In order to avoid confusion in later definitions, we use $\lambda \xrightarrow{SDG} \lambda'$ to denote any of the edges in the SDG and $\xrightarrow{SDG} *$ to denote a path.

4.3 Relevant Components

In order to construct the slice of the timed automata $\mathcal{A}$, we are required to obtain a set of relevant components i.e. locations, edges, update statements, guards and invariants from the original model, which is then used to construct the sliced timed automata $\mathcal{A}'$ (see Section 4.5).

To clarify, relevant components is a term used to describe components which are relevant with respect to some criteria, which we refer to as the slicing criteria. The slicing criteria is computed by extracting information concerning locations, clocks

and discrete variables from the CTL formula φ to be verified. From Section 3.2, it should be clear that CTL formulae may contain locations, clocks and discrete variables since these constitute the atomic propositions as given in Definition 2.3.4 of the timed automata model. As in the previous Section, we shall use the term variables to denote both clocks and discrete variables.

Definition 11. *(Slicing criteria)*
For some CTL formula φ, let P^{φ} be the set of atomic propositions of φ and let $vars(P^{\varphi})$ denote the set of variables which appear in the propositions of P^{φ}. The *slicing criteria* for a network of timed automata $(\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n)$ with respect to a set of atomic propositions P^{φ} is a set of statements $\Lambda_{\varphi}(\mathcal{A})$ such that:

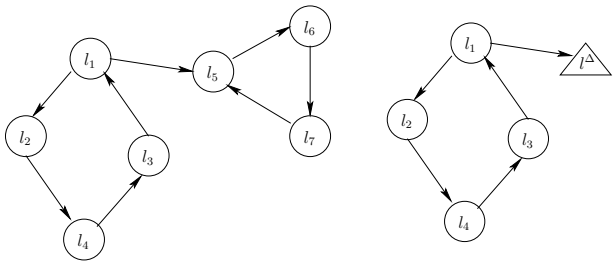
$$\Lambda_{\varphi}(\mathcal{A}) = \{\lambda \in \Lambda(\mathcal{A}) \mid def(\lambda) \cap vars(P^{\varphi}) \neq \emptyset\}$$

Relevant Automata Components

In [37] Janowska and Janowski propose to determine relevant locations of the automata, in terms of dependency relations. They use a notion of *control-dependency* and *time-dependency*. Control-dependencies occur whenever a set of locations in the automata may influence the control-flow such that other locations are bypassed and never reached. If a relevant location is control-dependent on another, the controlling location is consequently also relevant. The time-dependency relation contains pairs of locations for which there exists a path from one to the other, such that the time at which the latter is reached is influenced by the first. That is, the invariant and guards of the location introducing the delay does not conspire to force immediate progress. Any locations introducing such delays for relevant locations are as such also relevant.

The approach presented here does not attempt to compute any such dependency relations. Instead, we will focus only on slicing (away) locations which are irrelevant with respect reachability properties. Thus we propose that a set of irrelevant locations $L_{ir} \subseteq L$ must be supplied by the user or some tool, computed by some other approach, alternatively all locations are considered relevant, that is $L_r = L \setminus L_{ir}$. This requirement is introduced in order to simplify the technique presented here, which focuses on slicing the imperative elements of the model i.e. variables and statements.

To handle irrelevant locations, we propose to introduce a single sink location (with no outgoing edges) to which edges targeting all irrelevant locations are redirected. If an edge connects two irrelevant locations the edge is simply removed. Figure 4.8 illustrates the sink location as a triangle named l^{Δ}.
It is obvious that, when considering reachability properties, some models are constructed in such a way that a subset of its locations may easily be determined to be irrelevant when considering a specific reachability property. Assume, for example, that the property which we would like to verify for the model in Figure 4.8 involves reaching l_4. At no point in time from l_5 can l_4 ever be reached. Thus, it is obvious that l_5, l_6 and l_7 are irrelevant and $E\Diamond l_4$ holds for both 4.8(a) and 4.8(b) whereas $A\Diamond l_4$ holds for neither.



(a) Example Automata (b) Automata with sink

Figure 4.8: Sink location

Relevant Statements

The set of statements which should be included in the sliced automata may be computed as the transitive traversal of the nodes in the SDG[2] constructed from the automata. The traversal should of course originate from the set of statements and guards in the slicing criteria $\Lambda_{\varphi}(\mathcal{A})$.

4.4 Slicing Algorithms

In Section 4.5, we introduce the construction of the slices. This construction is based on a set of relevant locations and a set of relevant statements $\Lambda'(\mathcal{A})$. This Section presents two algorithms which will compute $\Lambda'(\mathcal{A})$. First we will show in Algorithm 1 how one, without considering summary edges, may compute the transitive closure of the SDG to achieve $\Lambda'(\mathcal{A})$.

4.4.1 Standard Algorithm

The approach taken in the following algorithm is based on Weisers original approach for slicing [52]. Although it presents nicely as an intuition, we shall later extend this approach in such a way that it avoids the *calling context problem* (Section 4.1.1).

Algorithm 1 Standard Algorithm for computation of relevant statements

$G = \bigcup_{l \in L_r} guard(l)$ the list of guards g
$WL =$ list of statements initially containing $\Lambda_{\varphi}(\mathcal{A})$
$WL = WL \cup \{\lambda \in \Lambda(\mathcal{A}) | \exists g \in G : \lambda \xrightarrow{sdg} g\}$
while $WL \neq \emptyset$ **do**
  take λ from WL and insert λ into $\Lambda'(\mathcal{A})$
  add to WL all $\lambda' \in \Lambda(\mathcal{A}) \mid \lambda' \xrightarrow{sdg} \lambda$
end while

[2] Although it is not within the intended scope of this thesis to present or highlight optimizations, it is worth pointing out that the task of constructing the SDG need only be undertaken once, only the simple computation of the set of relevant components must be computed for each verification task.

Algorithm 1 is based on a simple worklist approach where the worklist is initialized with the slicing criteria and all statements which reaches a guard on an outgoing edge from a relevant location. Based on this set, the transitive closure of the dependencies in the SDG is computed.

4.4.2 Improved Algorithm

Exploiting the previously introduced summary edges, Algorithm 2 will compute $\Lambda'(\mathcal{A})$ in such a way that edges are not followed into irrelevant *call-sites*. We use the notation $typeof(\xrightarrow{sdg})$ to obtain the type of the SDG edge. Intuitively the algorithm computes the same transitive closure as Algorithm 1, but it does so, by initially ignoring all *actual-out-edges* in the SDG. Having "collected" all statements reachable, the procedure is re-run, this time ignoring all *actual-in-edges*.

Algorithm 2 Precise computation of relevant statements

$G = \bigcup_{l \in L_r} guard(l)$ the list of guards g

prodcedure helper(WL : λ set, $\mathbb{E}$: set of SDG edge types, RESULT : λ set)
while $WL \neq \emptyset$ **do**
  take λ from WL and insert λ into RESULT
  for all edges $\lambda' \xrightarrow{sdg} \lambda$ where $\lambda' \in \Lambda(\mathcal{A})$ and $typeof(\xrightarrow{sdg}) \notin \mathbb{E}$ **do**
    add λ' to WL
  end for
end while
end prodcedure

TMP = $\emptyset$
INIT = $\Lambda_\varphi(\mathcal{A}) \cup \{\lambda \in \Lambda(\mathcal{A}) | \exists g \in G : \lambda \xrightarrow{sdg} g\}$
call helper(INIT, { *actual-out-edges* }, TMP)
call helper(TMP, { *actual-in-edges* }, $\Lambda'(\mathcal{A})$)

Algorithm 2 employes the help of a procedure called *helper* which takes as parameters; a set of statements to be used as a worklist (WL), a set of SDG edge types ($\mathbb{E}$) which should not be traversed and a result set (RESULT), which ultimately is to contain all of the statements which are found relevant by the helper procedure. The algorithm calls *helper* twice to compute $\Lambda'(\mathcal{A})$. The first call to the algorithm uses the set $\Lambda_\varphi(\mathcal{A})$ and all statements λ which reaches a guard $g \in G$ as the initial worklist, which is then added to RESULT, as the transitive traversal (of all but *actual-out-edges*) is computed. Since we need to continue the traversal from the set of statements found relevant in the first call, we use TMP to temporary hold this set. The contents of the temporary set is then copied to $\Lambda'(\mathcal{A})$ as the algorithm traverse the SDG a second time, using all but the *actual-out-edges*.

Notice that Algorithm 2 is equivalent to Algorithm 1 if the helper procedure is called only once with an empty set of SDG edges to reject: *call helper(*$\Lambda_\varphi(\mathcal{A}), \{\}, \Lambda'(\mathcal{A})$ *)*

The Calling Context Problem in SDG Terms

In the following we illustrate, using simplified FDGs, how the calling context problem is present using algorithm 1 and how it is avoided using Algorithm 2.

Figure 4.9 shows an SDG composed of five FDGs, where *actual-in-edges* (dotted) and *actual-out-edges* (solid) are connecting the the sub graphs. *Call-edges* are left out, as they do not add to the example.

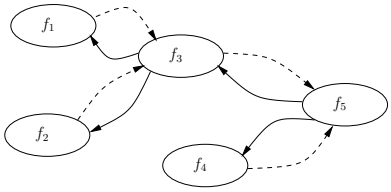


Figure 4.9: Simplified SDG exhibiting the calling context problem.

We illustrate (see Figure 4.10) the calling context problem using the case where the FDG for f_3 contains a relevant assignment $v = p_1 * f_5()$, for which the SDG vertex has an incoming *data-flow* edge from the *initialization* vertex for parameter p_1 and a *actual-in-edge* representing a call to f_5. Furthermore, Figure 4.10 shows the incoming *actual-in-edges* from functions f_1 and f_2 assigning p_1 with an actual value. The *actual-out-edges* are left out.

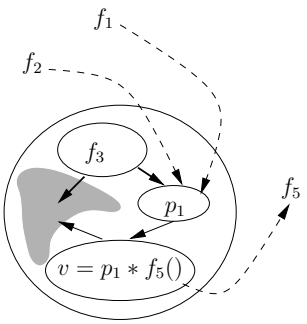


Figure 4.10: Illustrates the FDG for function f_3

When the slicing criteria initially contains $v = p_1 * f_5()$ in the FDG for f_3 the transitive closure computed by Algiorithm 1 reaches all FDGs in Figure 4.9. Using Algorithm 2 the resulting traversal uses only the edges shown in Figure 4.11 and reaches only the FDGs for f_1, f_2, f_3, f_5 patching the traversal using the summary edges.

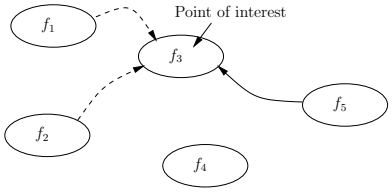


Figure 4.11: Illustrates the edges traversed in Figure 4.9 using Algorithm 2 given the point of interest.

4.5 The Slice

This Section presents the construction of the slice. We use the set $\Lambda'(\mathcal{A})$ to denote the set of statements which has been found relevant (in Section 4.4 we present two algorithms which will compute this set). Construction is introduced in two steps: First we present our approach for constructing the imperative elements i.e. updates and functions and second, we show how the sliced automaton is constructed.

4.5.1 Constructing Sliced Imperative Components

In this Section and the remaining chapters of this thesis, we use the notation $\overline{\lambda}$ to denote the sliced statement corresponding to λ.

Since an update $\alpha\beta$ is expressed over the syntactic productions `act_seq` and `asg_seq` these are simply (possibly empty) sequences of assignment and function call statements. Informally we say that a sliced version of an update is a same length sequence of statements where all statements not included in $\Lambda'(\mathcal{A})$ are substituted by the `skip` statement. Moreover, all statements are included in the original order.

Definition 12. *(Constructing Sliced Assignments and Actions)*
Let λ_{asg} be a single clock assignment in $Asg(\mathcal{C})$ and let λ_{act} be a single action i.e. assignment or function call in $Act(\mathcal{V}, F)$, then the corresponding sliced statements are defined such that:

$$\overline{\lambda_{asg}} = \begin{cases} \lambda_{asg} \text{ if } \lambda_{asg} \in \Lambda'(\mathcal{A}) \\ otherwise \ \texttt{skip} \end{cases} \qquad \overline{\lambda_{act}} = \begin{cases} \lambda_{act} \text{ if } \lambda_{act} \in \Lambda'(\mathcal{A}) \\ otherwise \ \texttt{skip} \end{cases}$$

Note that for a statement λ on the form `skip`, $\overline{\lambda} = \lambda$.

Definition 13. *(Statement Sequences)*
Let $\lambda_{seq} = \lambda_1 \ \lambda_2 \ \ldots \lambda_n$ for $n > 1$ be a non-empty sequence of statements occurring in $\mathcal{A}$. Then the corresponding slice is a sequence $\overline{\lambda_{seq}} = \overline{\lambda_1} \ \overline{\lambda_2} \ \ldots \overline{\lambda_n}$.

Definition 14. *(Update Construction)*
Let $\alpha\beta$ be an update for an edge e, then $\overline{\alpha\beta}$ is the sliced update for the edge e' in the sliced model (see Definition 17). Since α and β are sequences, both $\overline{\alpha}$ and $\overline{\beta}$ are sequences, constructed such that they satisfy the properties of Definition 13. Moreover, since all statements in $\alpha\beta$ are either variable actions or clock assignments, the construction of $\overline{\lambda} : \lambda \in \Lambda(\alpha\beta)$ is given by Defintion 12.

The construction of sliced functions are very similar to that of updates. Functions essentially consists of sequences of statements possibly controlled by an *if* or *while* statement.

Definition 15. *(Constructing Sliced while, if and return Statements)*
Let λ_{while} and λ_{if} in $\Lambda(\mathcal{A})$ be occurrences of *while* and *if* statements in the timed automata $\mathcal{A}$. The corresponding sliced statements are constructed such that:

- For a statement λ_{while} on the form `while(`φ`){` $\lambda_1\ \lambda_2\ \ldots \lambda_n$ `}`,
  $\overline{\lambda_{while}} = \begin{cases} \texttt{while(}\varphi\texttt{)\{}\overline{\lambda_1}\ \overline{\lambda_2}\ \ldots \overline{\lambda_n}\texttt{\}} \text{ if } \lambda_{while} \in \Lambda'(\mathcal{A}) \\ \textit{otherwise } \texttt{skip} \end{cases}$
- Dually, for a statement λ_{if} on the form `if(`φ`){` $\lambda_1\ \lambda_2\ \ldots \lambda_n$ `}`
  $\overline{\lambda_{if}} = \begin{cases} \texttt{if(}\varphi\texttt{)\{}\overline{\lambda_1}\ \overline{\lambda_2}\ \ldots \overline{\lambda_n}\texttt{\}} \text{ if } \lambda_{if} \in \Lambda'(\mathcal{A}) \\ \textit{otherwise } \texttt{skip} \end{cases}$
- For a statement λ_{return} on the form `return` *expr*
  $\overline{\lambda_{return}} = \begin{cases} \texttt{return } \textit{expr} \text{ if } \lambda_{return} \in \Lambda'(\mathcal{A}) \\ \textit{otherwise } \texttt{skip} \end{cases}$

Definition 16. *(Function Construction)*
Let f be a function belonging to the timed automata $\mathcal{A}$ and let $\overline{f}$ denote the slice of f constructed from $\Lambda'(\mathcal{A})$. Given the syntax of a function, the body of a function is simply a sequence of statements over `stmt_seq`, thus we define $\overline{f}$ as f where the occurring sequence of statements $\lambda_1\ \lambda_2\ \ldots \lambda_n$ is replaced by $\overline{\lambda_1}\ \overline{\lambda_2}\ \ldots \overline{\lambda_n}$.

4.5.2 Constructing the Sliced Timed Automata

In the following definition we use L_r to denote the set of relevant locations. Moreover, we use $guards(l)$ to obtain all guards of outgoing edges from location l. Formally: $guards(l) = \bigcup_{e \in out(l)} guard(e)$.

Definition 17. *(Automata Slicing)*
For a network of n timed automata $(\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n)$ where $\mathcal{A}_i = \langle L_i, l_i^0, \mathcal{V}_i, \mathcal{C}, \Sigma, F, E_i, I_i \rangle$ for $1 \leq i \leq n$ and a variable valuation $\omega_0 : \mathcal{V} \to \mathbb{Z}$ where $\mathcal{V} = \bigcup_{1 \leq i \leq n} \mathcal{V}_i$. A slice, with respect to P^{φ} is a network of timed automata $(\mathcal{A}'_1 \parallel \ldots \parallel \mathcal{A}'_n)$ where $\mathcal{A}' = \langle L'_i, l'^0_i, \mathcal{V}'_i, \mathcal{C}', \Sigma', F', E'_i, I'_i \rangle$ for $1 \leq i \leq n$ with a variable valuation $\omega'_0 : \mathcal{V}' \to \mathbb{Z}$ where $\mathcal{V}' = \bigcup_{1 \leq i \leq n} \mathcal{V}'_i$ and $\omega'_0(\mathcal{V}') = \omega_0(\mathcal{V}')$ and for each $\mathcal{A}_i$ such that $L_i \cap L'_i \neq \emptyset$ the following holds:

- $L'_i = L_i \cap L_r \cup l_i^{\Delta}$
- $l'^i_0 = l^i_0$ since l^i_0 must always be considered relevant.
- $\mathcal{V}'_i = \mathcal{V}_i \cap \bigcup_{\lambda \in \Lambda'(\mathcal{A})} vars(\lambda)$
- $\mathcal{C}'_i = \mathcal{C}_i \cap \left(\bigcup_{\lambda \in \Lambda'(\mathcal{A})} vars(\lambda) \cup \bigcup_{l \in L_r} \Big(I(l) \cup guards(l) \Big) \right)$
- E'_i is the smallest set such that for each $e \in out(l) \mid l \in L'_i$ there exists an $e' \in E'_i$, where:

$source(e') = source(e)$,
$guard(e') = guard(e)$,
$label(e') = label(e)$,
$update(e') = \overline{\alpha\ \beta}$ where $\alpha\beta = update(e)$

$$target(e') = \begin{cases} target(e) \text{ if } target(e) \in L'_i, otherwise \\ l_i^{\Delta} \text{ where } l^{\Delta} \text{ is described in Section 4.3} \end{cases}$$

- $\Sigma'_i = \bigcup_{e \in E'_i} label(e')$
- $F' = \bigcup_{f \in F} \overline{f}$
- I'_i is a function $I'_i : L'_i \to \Phi(\mathcal{V}) \times \Psi(\mathcal{C})$ where $\forall l \in L'_i : I'_i(l) = I(l)$

4.6 Correctness

Here we introduce a notion of correctness for our slicing approach and we prove that the approach presented does in-fact create correct syntactic reductions of the extended notion of timed automata introduced. In the following definitions and proofs we show that the slicing approach preserves reachability properties.

We choose first to prove that whenever sliced network of timed automata is constructed using the approach in Definition 17, it is related to the original network of timed automata in such a way that they satisfy exactly the same reachability properties. We then proceed to prove that the algorithm, which determines which statements to include and the construction of sliced imperative components (from Section 4.5.1), does produce a slice satisfying the reachability property exactly when the original does.

4.6.1 $E\Diamond\varphi$-Bisimulation and Reachability

To prove correctness of the construction in Definiton 17, we introduce the notion of an $E\Diamond\varphi$-Bisimulation R_φ and the induced equivalence relation $\equiv_\varphi$. Theorem 19, ensures that an $E\Diamond\varphi$-Bisimulation is indeed reachability preserving. Proving correctness is then reduced to proving Lemma 21 which says that the relation $\simeq$, which relates a network of timed automata $\mathcal{A}$ with its slice $\mathcal{A}'$, is indeed an $E\Diamond\varphi$-Bisimulation.

Definition 18. *($E\Diamond\varphi$-Bisimulation)*
A relation R_φ is a $E\Diamond\varphi$-Bisimulation if, whenever $s_1 R_\varphi s_2$ then either

1. $s_1 \not\models E\Diamond\varphi \wedge s_2 \not\models E\Diamond\varphi$
   or
2. **x:** $s_1 \to s'_1 \Rightarrow \exists s'_2 : s_2 \to s'_2 \wedge s'_1 R_\varphi s'_2$
   y: $s_2 \to s'_2 \Rightarrow \exists s'_1 : s_1 \to s'_1 \wedge s'_1 R_\varphi s'_2$
   z: $s_1 \models \varphi$ iff $s_2 \models \varphi$

We write $s_1 \equiv_\varphi s_2$ if $s_1 R_\varphi s_2$ for some $E\Diamond\varphi$-Bisimulation R_φ.

The following result says that whenever two states are related by an $E\Diamond\varphi$-Bisimulation they agree on the reachability of φ.

Theorem 19. If $s\ R_\varphi\ s'$ for some $E\Diamond\varphi$-Bisimulation then:

$$s \models E\Diamond\varphi \iff s' \models E\Diamond\varphi$$

Proof of theorem 19. It suffices to prove $\Rightarrow$ as whenever R_φ is an $E\Diamond\varphi$ Bisimulation then so is its transpose R_φ^{-1}. We prove by induction in n that whenever $sR_\varphi s'$ then the *Induction Hypothesis (IH)*:

$$\begin{array}{c}\forall n \geq 0 : s \rightarrow^n t \text{ where } t \models \varphi \Rightarrow \\ \exists t' : s' \rightarrow^n t' \text{ with } t \models \varphi\end{array}$$

holds.

Basis: $n = 0$, then $s \models \varphi$ and by **z** in Definition 18 also $s' \models \varphi$

The Inductive Step: Assume $s \rightarrow p \rightarrow^n t$ with $t \models \varphi$. Since $sR_\varphi s'$, $s' \rightarrow p'$ with $pR_\varphi p'$ by Definition 18. Now using the *IH* it follows that $p' \rightarrow^n t'$ for some t' where $t' \models \varphi$, and hence $s' \rightarrow^{n+1} t'$.

□

The $\simeq$ Relation

The relation introduced in (the following) Definition 20 relates the transition system $\mathcal{S}$ for a timed automata network $\mathcal{A}$ and the transition system $\mathcal{S}'$ for a timed automata network $\mathcal{A}'$ whenever $\mathcal{A}'$ is a slice constructed from $\mathcal{A}$ as defined in Definition 17. In Definition 20 we write Θ to denote the set of locations which easily can be computed never to satisfy a formula φ. Furthermore, we use l^Δ to denote a sink location in the sliced model.

Definition 20. *(The $\simeq$ Relation)*
Let $s = ((l_1, \ldots, l_n), \omega, \sigma) \in S, s' = ((l'_1, \ldots, l'_n), \omega', \sigma') \in S'$ and $\simeq \subseteq S \times S'$. We say that the state s is related to the state s' (we write $s \simeq s'$) iff:

a) $\exists i : l'_i = l^\Delta \wedge l_i \in \Theta$, or
b) $\forall i : l_i = l'_i \wedge \omega(\mathcal{V}') = \omega'(\mathcal{V}') \wedge \sigma(\mathcal{C}') = \sigma'(\mathcal{C}')$

The relation $\simeq$ is a $E\Diamond\varphi$-Bisimulation

The proof of Lemma 21 confirms that the slicing presented in this thesis does in-fact preserve reachability properties, whenever the claim in the supporting Lemma 25 holds.

Lemma 21. The relation $\simeq \subseteq S \times S'$ is a $E\Diamond\varphi$-Bisimulation between two structures $M = (S, \mathcal{I})$ and $M' = (S', \mathcal{I}')$.

Proof of Lemma 21: For $s = ((l_1, \ldots, l_n), \omega\sigma) \in S$ and $s' = ((l'_1, \ldots, l'_n), \omega', \sigma') \in S'$ where $s \simeq s'$

- if a) in Definition 20 holds, then 1. in Definition 18 holds trivially

- if b) in Definition 20 holds, then we must show that 2. in Definition 18 holds
  - **2x:** The proof is in two parts; Where s either *delays* or take a *discrete* transition. We show that the transition may match by s'
    - ***2x1 (delay):*** $s \xrightarrow{d} t$ where $t = ((l_1, \ldots, l_n), \omega, \sigma')$. Because b) holds, we know that s and s' have the same location vectores and hereby also the same invariants to satisfy. Thus, the same delay d is possible at s' such that $s' \xrightarrow{d} t'$ where $t' = ((l_1, \ldots, l_n), \omega, \sigma')$. We now have to show that $t \simeq t'$:

      It is trivial that $\forall i : l_i = l'_i$ and in the case of delays; $\omega = \omega'$ must hold. Finally, if $\sigma = \sigma'$ then $\sigma(d) = \sigma'(d)$ given the semantics on page 29, thus $t \simeq t'$.

    - ***2x2 (discrete):*** $s \xrightarrow{a} t$ where $t = ((p_1, \ldots, p_n), \upsilon, \rho)$ where $e = l_i \xrightarrow{\psi\varphi\ a\ \beta\alpha} m_i$ is the edge taken, means that:

      - $guard(e)$ is satisfied: $\omega, \sigma \models \psi$ and $\omega \models \varphi$,
      - The update is evaluated as: $[\![\alpha]\!](\omega) = \upsilon$, $[\![\beta]\!](\upsilon, \sigma) = \rho$
      - and for the location vector : $p_i = m_i \wedge \forall j \neq i : l_j = p_j$.

      $s' \xrightarrow{a} t'$ by taking e' where $t' = ((p'_1, \ldots, p'_n), \upsilon', \rho')$, and e' is the same as e but with β and α sliced. Because b) holds s' must satisfy the guards: $\omega', \sigma' \models \psi$ and $\omega' \models \varphi$. Since the edge taken is the same as before but with β and α sliced, denoted as $\overline{\beta}$ and $\overline{\alpha}$ respectively. This means that:

      - The update is evaluated as: $[\![\overline{\alpha}]\!](\omega') = \upsilon', [\![\overline{\beta}]\!](\upsilon', \sigma') = \rho'$,
      - and for the location vector $p'_i = m_i \wedge \forall j \neq i : l_j = p_j$.

      We must show that $t \simeq t'$ for two cases:

      - $m_i \notin \Theta$ which means b) must hold: It is trivial that $\forall i : p_i = p'_i$ and we know from lemma 25 that $(\rho', \upsilon') \equiv_{p_{m_i}} (\rho, \upsilon)$ where p_{m_i} is the program point at m_i.
      - $m_i \in \Theta$ which means a) and holds trivially.

  - **2z:** The proof that $s \models \varphi$ iff $s' \models \varphi$ holds, follows from the fact that, whenever b) holds we know that $\forall i : l_i = l'_i$ and $\mathcal{V}'$ and $\mathcal{C}'$ obviously contain all variables and clocks from φ (given Definition 11). Furthermore, we know from Lemma 24 (see page 64) that $[\![expr]\!](\sigma', \omega') = [\![expr]\!](\sigma, \omega)$

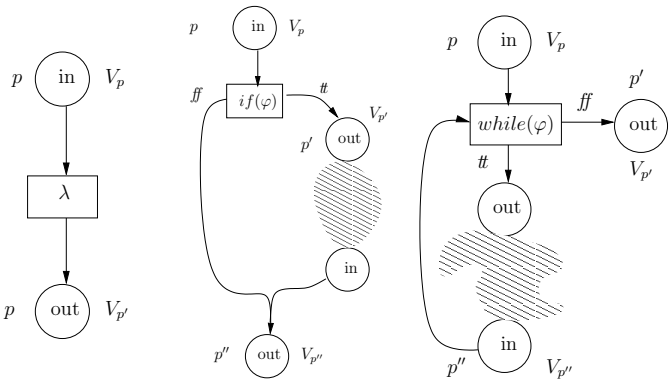
□

Whenever the set of irrelevant locations is empty it is trivially true that case *a* of Definition 20 is never satisfied. Thus, all states related by $\simeq$ must be equal with respect to the locations and the clock- and variable valuations. Note that

the relation $\simeq \subseteq S \times S'$ is a bisimulation between two structures $M = (S, \mathcal{I})$ and $M' = (S', \mathcal{I}')$ if $L_{ir} = \emptyset$.

4.6.2 Proving The Construction of Imperative Components

The remaining task is concerned with proving that Lemma 25 and Lemma 24 holds when imperative constructs of a network of extended timed automata are sliced as defined in Section 4.5.1 using Algorihm 1. We proceed to show this in the following way: We show that there exists a static analysis which, given a starting point (program point) and a set of initially relevant variables, may be used to annotate each program point in the control-flow graph of an imperative code fragment with the a set of relevant variables (see Figure 4.12). We continue to prove that having obtained this annotation, all assignments which assign a variable not contained within the annotated set, may be replaced by `skip` without effecting the the value of the initially relevant variables. Having proven this, we move on to show that the approach introduced in this thesis, for slicing imperative components, will produce the exact same slice. That is, whenever it can be proven using the static analysis that a statement must be preserved in the slice, algorithm 1 will include it in the set of relevant statements $\Lambda'(\mathcal{A})$.



(a) assignment (b) if (c) while

Figure 4.12: Control-flow of imperative constructs annotated with program points and set of locally relevant variables

Calculating Relevant Variables

The following static analysis form a constraint system on the set of relevant variables at each program point in the CFG (see Section 4.2.2) for a fragment of imperative code expressed in the syntax on page 23.

In the following definition, we use the function $V_{rel}(\cdot)$ to denote the set of variables $(\mathcal{V} \cup \mathcal{C})$ which are relevant with respect to expressions in `expr` and boolean expressions in `bexp`. The function is defined inductively over expressions such that:

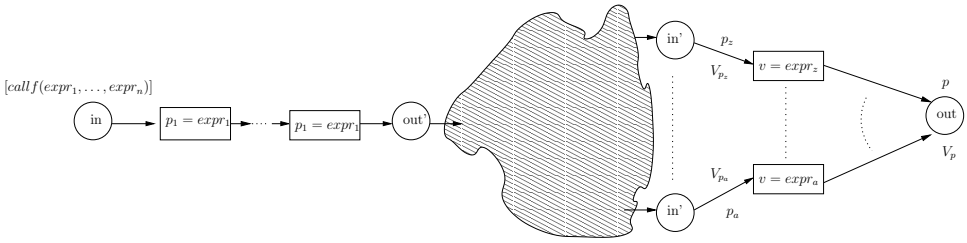


Figure 4.13: Annotated in-lined function

$$\begin{aligned} V_{rel}(\mathtt{m}) &= \emptyset \qquad V_{rel}(v) = \{v\} \\ V_{rel}(expr_1 \otimes expr_2) &= \mathcal{V}_{rel}(expr_1) \cup V_{rel}(expr_2) \\ V_{rel}(-expr) &= V_{rel}(expr) \\ V_{rel}(funCall) &\text{ is given by the equivalent statement} \end{aligned}$$

The function $V_{rel}(\cdot)$ may trivially be extended to include boolean expressions in the same manner, thus this extension is omitted here.

Definition 22. *(Annotation Constraints)*
For each construction of the imperative language, we construct a constraint describing the set of variables V_p which have relevance (i.e. can be reached) before entering the construct (see Figure 4.12).

For *skip* statements the constraint is trivial:

$$V_p \supseteq V_{p'}$$

For assignments (e.g. $v = expr$) the constraint is :

$$\begin{gathered} V_p \supseteq V_{p'} \setminus \{v\} \cup V_{rel}(expr) \text{ if } v \in V_{p'} \\ \text{otherwise } V_p \supseteq V_{p'} \end{gathered}$$

For *if* and *while* statements (assuming the condition φ) the constraint is:

$$V_p \supseteq V_{p'} \cup V_{p''} \cup V_{rel}(\varphi)$$

For *return* statements (assuming the expression $expr$) the constraint is:

$$V_p \supseteq V_{p'} \cup V_{rel}(expr)$$

For function calls with parameters $expr_1, \ldots expr_n$ the constraint is:

$$V_p \supseteq \left(V_{p''} \cup V_{p'}\right) \cup \bigcup_{i \in \{1,\ldots,n\}} V_{rel}(expr)$$

Note that the constraints express a monotone function $\mathrm{F}_V{:}(P \mapsto 2^V) \mapsto (P \mapsto 2^V)$, thus the set V_p for any program point p may be computed as a least fix-point. Let P be the set of program points. Consider a mapping $(P \mapsto 2^V)$ i.e. for each program point a set of variables is assigned. We order mappings $(P \mapsto 2^V)$ by $\mathrm{f} \sqsubseteq \mathrm{g}$ iff $\forall p \in P : \mathrm{f}(p) \subseteq \mathrm{g}(p)$. Then $\mathrm{F}_V{:}(P \mapsto 2^V) \mapsto (P \mapsto 2^V)$ is monotone.

Using the previous constraint system to decorate each program point p with a set V_p, the following Lemma (25) ensures that variable and clock valuations, at the same program point of an imperative code fragment and its slice, are equal with respect to the set V_p.

Definition 23. *(Variable and Clock Valuation Equivalence)*
Let (σ, ω) and (σ', ω') be arbitrary pairs of clock and variable valuations and let $\mathcal{V}_p$ and $\mathcal{C}_p$ be the sets of relevant variables and clocks at program point p. We say that $(\sigma, \omega) \equiv_p (\sigma', \omega')$ whenever:

- $\forall u \in \mathcal{C}_p : (\sigma, \omega)(u) = (\sigma', \omega')(u)$
- $\forall v \in \mathcal{V}_p : (\sigma, \omega)(v) = (\sigma', \omega')(v)$

Lemma 24. *(Expression Evaluation in $(\sigma_1, \omega_1) \equiv_p (\sigma_2, \omega_2)$)*
For two valuations (σ_1, ω_1) and (σ_2', ω_2'), we have that $[\![expr]\!](\sigma_1, \omega_1) = [\![expr]\!](\sigma_2, \omega_2)$ whenever:

$(\sigma_1, \omega_1) \equiv_p (\sigma_2, \omega_2)$ and

$vars(expr) \subseteq V_p$

Proof of Lemma 24. We prove Lemma 24, by induction in the structure of the expression syntax (see Section 2.3.1).

IH: $\forall\ expr \in Expr(\mathcal{V}, F)$ where $vars(expr) \subseteq V_p : [\![expr]\!](\omega_1, \sigma_1) = [\![expr]\!](\omega_2, \sigma_2)$ whenever $(\omega_1, \sigma_1) \equiv_p (\omega_2, \sigma_2)$

Basis:
Given the semantics of $[\![m]\!]$ it must always be the case that $[\![m]\!](\sigma, \omega)_a = [\![m]\!](\sigma, \omega)_b$. Since $(\omega, \sigma)_1 \equiv_p (\omega, \sigma)_2$ we have by definition that:

$\forall v \in \mathcal{V}_p : [\![v]\!](\sigma_1, \omega_1) = [\![v]\!](\sigma_2, \omega_2)$.
$\forall u \in \mathcal{C}_p : [\![u]\!](\sigma_1, \omega_1) = [\![u]\!](\sigma_2, \omega_2)$.

The Inductive Step:
Assume that the *IH* holds for expressions $expr_1$ and $expr_2$, we then have to prove that *IH* holds for $expr_1 \otimes expr_2$ where $\otimes \in \{-, +, *, /\}$.
Given the definition of the semantics, we have that $[\![expr_1 \otimes expr_2]\!](\sigma_1, \omega_1) = (val, \sigma_1, \omega_1')$ iff $\langle expr_1 \otimes expr_2, \sigma_1, \omega_1\rangle \rightarrow \langle val, \sigma_1, \omega_1'\rangle$ where $val = val_1 \otimes val_2$ and val_1 is given by $[\![expr_1]\!](\sigma_1, \omega_1)$ just as val_2 is given by $[\![expr_2]\!](\sigma_1, \omega_1'')$. Now using the *IH* it follows that $[\![expr_1]\!](\sigma_1, \omega_1,) = [\![expr_1]\!](\sigma_2, \omega_2) = val_1$ and $[\![expr_2]\!](\sigma_1, \omega_1'') = [\![expr_2]\!](\sigma_2, \omega_2'') = val_2$, hence $[\![expr_1 \otimes expr_2]\!](\sigma_2, \omega_2) = (val, \sigma_2, \omega_2')$ where $val = val_1 \otimes val_2$. □

Lemma 25. Let $(\sigma, \omega)_n$ (and $(\sigma, \omega)_n$) be the clock and variable valuation in $\mathcal{A}$ (and $\mathcal{A}'$) after n steps at program point p (and p'), then:

$$(\sigma, \omega) \equiv_p (\sigma', \omega')$$

We continue to prove Lemma 25 in the presence of the CFG annotations computed using the static analysis. Also the following proof assumes that the structure of the control-flow graph remains unaltered, such that any program point in the original control-flow exists in the slice.

Proof of Lemma 25 (based on CFG annotations). Assume for any control-flow graph G and its slice $\overline{G}$ that $(\sigma,\omega)_0 \equiv_{p_0} (\sigma',\omega')_0$ at first program point p_0 in both G and $\overline{G}$. We show that Lemma 25 holds by proof of induction in the number of program steps.

IH: Let $(\sigma,\omega)_n$ and $(\sigma',\omega')_n$ be the valuations after n steps of the program execution and assume that:

1. $(\sigma,\omega)_n$ and $(\sigma',\omega')_n$ will occur at exactly the same program point p_n
2. and $(\sigma,\omega)_n \equiv_{p_n} (\sigma',\omega')_n$

Base: $(n = 0)$ after 0 steps, $(\sigma,\omega)_n \equiv_{p_n} (\sigma',\omega')_n$ holds trivially by the initial assumptions that $(\sigma,\omega)_0 \equiv_{p_0} (\sigma'_0,\omega')_0$.

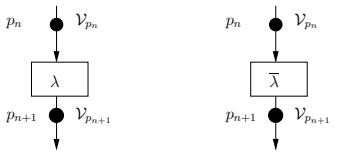


Figure 4.14: Illustrating two statements from G and $\overline{G}$ at the same program point

The Inductive Step: Assuming that the *IH* holds after n steps, we use the annotations (presented in Definition 22) to prove that the IH holds at p_{n+1}. (Figure 4.14 illustrates the program points and their respective sets of locally relevant variables.)

Let λ_n be the assignment $v = expr$, it must then be the case that a) $v \in \mathcal{V}_{p_n}$ or b) $v \notin \mathcal{V}_{p_n}$ and whenever a) holds $\overline{\lambda}_n \equiv v = expr$ and when b) holds $\overline{\lambda}_n \equiv \texttt{skip}$.

Case a) $\mathcal{V}_{p_n} \supseteq (\mathcal{V}_{p_{n+1}} \setminus \{v\}) \cup \mathcal{V}_{rel}(expr)$ and:

$$\begin{aligned}(\sigma,\omega)_{n+1} &= [\![v = expr]\!](\sigma,\omega)_n \\ &= (\sigma,\omega[v \mapsto val])_n \text{ where } val = [\![expr]\!](\sigma,\omega)_n \\ &= (\sigma,\omega[v \mapsto val])_n \text{ where } val = [\![expr]\!](\sigma',\omega')_n \text{ (by Lemma 24)}\end{aligned}$$

$$(\sigma',\omega')_{n+1} = [\![v = expr]\!](\sigma',\omega')_n$$

We must now show that $(\sigma,\omega)_{n+1} \equiv_{p_{n+1}} (\sigma',\omega')_{n+1}$. That is, $\forall v' \in \mathcal{V}_{p_{n+1}}$: $(\sigma,\omega)_{n+1}(v') = (\sigma',\omega')_{n+1}(v')$. This yields two further cases:

- $(v \neq v')$: then $v' \in \mathcal{V}_{p_n}$ and $(\sigma,\omega)_{n+1}(v') = (\sigma\omega)_n(v')$ and $(\sigma',\omega')_{n+1}(v') = (\sigma',\omega')_n(v')$. By *IH* we know that $(\sigma,\omega)_n \equiv_{p_n} (\sigma',\omega')_n$ thus $(\sigma,\omega)_{n+1}(v') = (\sigma',\omega')_{n+1}(v')$.
- $(v = v')$: then $(\sigma,\omega)_{n+1}(v) = [\![expr]\!](\sigma',\omega')_n = (\sigma',\omega')_{n+1}(v)$.

Case b) $\mathcal{V}_{p_{n+1}} \subseteq \mathcal{V}_{p_n}$ and:

$$\begin{aligned}(\sigma,\omega)_{n+1} &= [\![v = expr]\!](\sigma,\omega)_n \\ &= (\sigma,\omega[v \mapsto val])_n \text{ where } val = [\![expr]\!](\sigma,\omega)_n\end{aligned}$$

$$(\sigma',\omega')_{n+1} = [\![\texttt{skip}]\!](\sigma',\omega')_n = (\sigma',\omega')_n$$

We must now show that $(\sigma,\omega)_{n+1} \equiv_{p_{n+1}} (\sigma',\omega')_{n+1}$. That is, $\forall v' \in \mathcal{V}_{p_{n+1}}$: $(\sigma,\omega)_{n+1}(v') = (\sigma',\omega')_{n+1}(v')$. By the *IH* we have that $(\sigma,\omega)_n \equiv_{p_n} (\sigma',\omega')_n$ and since $v \notin \mathcal{V}_{p_n}$ and $\mathcal{V}_{p_{n+1}} \subseteq \mathcal{V}_{p_n}$ (given the constraints), it must be the case that $v \notin \mathcal{V}_{p_{n+1}}$. Hence, $(\sigma,\omega)_{n+1} \equiv_{p_{n+1}} (\sigma',\omega')_{n+1}$.

□

The above proof of Lemma 25 requires that the structure of the control-flow graph is preserved in order to reason about equivalence of variable valuations at program points. It is easily proven that after the static analysis has been used to slice assignments, we may, under the assumption that all conditional predicates are side-effect free, replace all control statements in the sliced CFG where the body contains only skip statements with one skip.

Lemma 26. *(Control Statement Exclusion)*
For any control statement λ with the body $\lambda_1\lambda_1 \ldots \lambda_n$, statement λ may be replaced with `ship` whenever $\bigwedge_{i\in\{1,\ldots,n\}} \lambda_i \equiv \texttt{skip}$

Proof. The proof is trivial, since $[\![\texttt{if}(\varphi)\{\texttt{skip}_1 \ldots \texttt{skip}_n\}]\!](\sigma,\omega) = [\![\texttt{skip}]\!](\sigma,\omega) = (\sigma,\omega)$ whenever φ is side-effect-free, for any $n \geq 1$. The same applies for *while*.

□

Combining CFG Annotation and Lemma 26

Removing assignments and control statements as defined in Lemma 26 in succession until no changes occur to the CFG will produce a slice preserving the valuation of variables at the initial program point of interest.

Annotation Matches Slicing Approach

Having proven that Lemma 25 holds in the presence of an annotated CFG, we must now show that the approach introduced in Section 4.5.1 is at-least as strong as CFG annotation.

Proving that the slicing algorithm produces a slice containing exactly the statements included by the static analysis (CFG annotation) and control statement exclusion (Definition 26), is done in two steps. First we show that whenever an assignment, assigns a variable which is locally relevant, it corresponds to the case where the assignment reaches a statement which has been determined relevant by algorithm 1 Intuitively reaching implies the existence of a *data-flow* edge in the SDG and the fact that an assignment has such an edge must imply its inclusion in the slice. The second step in the proof is to show that the *control-edges* are corresponds to the trivially proven technique in Definition 26.

Lemma 27. *(Reaching Contains at least the Annotation Information)*
Whenever $v \in \mathcal{V}_p$ where p is the program point immediately after λ on the form $v = expr$. Then λ reaches some statement λ' where $v \in ref(\lambda')$ and λ' is relevant.

(Proof of Lemma 27). For a statement λ to reach λ' we know from Definition 9 that two properties must be satisfied. 1) there must exist a control-flow path $\lambda \xrightarrow{cfg} *\lambda'$

and 2) the path must not contain a redefinition of the variable assigned in λ.

Proving that $v \in \mathcal{V}_p \Rightarrow \lambda \; reaches \; \lambda'$, where λ' is relevant requires two cases:

1) For a variable v to be relevant at program point p_i there must exist a relevant statement λ_j immediately before program point p_j where p_i occurs prior to p_j in the control-flow and $v \in ref(\lambda_j)$:

$$v \in \mathcal{V}_j \Rightarrow \exists \lambda_j : \xrightarrow{\mathcal{V}_{p_{j-1}}} \lambda_j \xrightarrow{\mathcal{V}_{p_j}} \text{ where } v \in \mathcal{V}_{p_{j-1}} \setminus \mathcal{V}_{p_j}$$

Assume for the purpose of reaching a contradiction that this is not the case (i.e. there exists no such λ_j) then by the annotation constraints $v \notin \mathcal{V}_{p_i}$ for any $\mathcal{V}_i$. which proves 1).

$$v \in \mathcal{V}_p \Rightarrow \exists \lambda' : \lambda \xrightarrow{cfg} *\lambda' \text{ where } \lambda' \text{ is relevant and } v \in ref(\lambda')$$

2) Since $v \in \mathcal{V}_p$ then there exists a control-flow path $\lambda \xrightarrow{cfg} *\lambda'$ such that $\forall \lambda''$ where $\lambda \xrightarrow{cfg} *\lambda'$ can be expressed as $\lambda \xrightarrow{cfg} *\lambda''$ and $\lambda'' \xrightarrow{cfg} *\lambda'$ the statement $\lambda'' \equiv v = expr'$, where $v \notin \mathcal{V}_{rel}(expr')$. Again, assume for the purpose of reaching a contradiction that there is only one control-flow path $\lambda \xrightarrow{cfg} *\lambda'$ and there existed a λ'', then at the program point $p'' - 1$ (immediately before λ'') $\mathcal{V}_{p''-1}$ cannot contain v. Since we know that $v \in \mathcal{V}_p$ then there must exist a path not containing λ''.

Since 1) and 2) holds, we know that whenever the static analysis from Definiton 22 finds a statement λ relevant, there exists a *data-flow* edge in the SDG from λ to a relevant statement λ' and by the transitive closure computed in Algorithm 1 statement λ is also included in $\Lambda'(\mathcal{A})$. □

Although we have now shown that our approach will indeed exclude assignments whenever the static analysis shows that the assignment may be skipped without effecting the valuation of relevant variables, it still remains to be proven that our approach only removes control statements which (as in Definition 26) does not have an effect on the valuation.

Lemma 28. *(Control Edges Correspond to Multiple Annotation Computations)* Given a control statement λ, it is included in $\Lambda'(\mathcal{A})$ by Algorithm 1 iff repeated program point annotation and control statement exclusion (Lemma 26) fails to exclude λ.

(Proof of Lemma 28). The proof of Lemma 28 is given by contraction:

($\Rightarrow$) Whenever a statement λ' in the body of a control statement (*if* or *while*) is not replaced by `skip`, λ' is relevant (given Lemma 25) and is included in $\Lambda'(\mathcal{A})$ (given Lemma 27). By definition of SDG and FDGs $\lambda \xrightarrow{sdg} \lambda'$ by a *control edge* and the transitive closure computed by Algorithm 1 will include λ.

($\Leftarrow$) Whenever all statements λ_i in the body of λ is replaced by `skip` - none are relevant, thus the algorithm cannot follow the before mentioned control edge.

□

Chapter

5

Implementation

This Chapter describes a prototype implementation for Uppaal of the slicing technique presented in Section 4. We introduce the existing parser library UTAP developed for Uppaal, which is used to create an *abstract-syntax-tree* of the input model.

Slicing is implemented for Uppaal as a preprocessing library, developed in C++, along with a stand-alone executable, which takes a model and a CTL query as input and outputs a sliced model, which can then be opened in Uppaal for verification.

Contents

5.1 Introduction

So far we have mainly been concerned with the theory of slicing models expressed in the extended timed automata formalism. In this Chapter we present a prototype implementation for Uppaal, which has been developed based on the slicing technique given in Chapter 4. It should be noted that the modeling language of Uppaal is somewhat larger than the one introduced in Chapter 2, making the implementation itself a testimony of the extendability of the theory.

The prototype, which we will refer to as UTASA (Uppaal Timed Automata Static Analyser) library, is developed as a preprocessing library for Uppaal, meaning that slicing takes place before any verification task. Furthermore, the intention is also to provide a library for other developers who wishes to do static analysis on Uppaal timed automata.

UTASA includes, besides the ability to slice, the following functionality:

- A data structure and a builder class to create a directed acyclic graph (DAG) [48] representation of the timed automata.
- Classes to transform the imperative C-like code of Uppaal into a subset of the well known SSA (Static Single Assignment) form [19]. Meaning that only a proof-of-concept normal form is implemented and not SSA in its full.
- A data structure and builder classes to create an SDG (System Dependency Graphs - see Section 4.2.3) representation of models.

Finally, the motivation behind a stand-alone library is also to encourage further development of static analysis and preprocessing tools for Uppaal.

Application

The implementation of UTASA described in the following Section has led to the development of a small tool in order to show its usage. Furthermore, the application was also developed to aid the testing described in Chapter 6.

The tool, called "TASlicer", takes as input a Uppaal timed automata XML file and a CTL expression and outputs to a provided destination the sliced version of the XML file. The sliced XML file can then be opened in Uppaal and model checking can commence. The advantage of the tool is, that by outputting to Uppaal timed automata XML, we are able to use the graphical user interface (GUI) provided with Uppaal and are therefore able to show the traces graphically. Finally, the visualization utility is hereby also available and users are able to see what goes on during execution of the sliced model.

As future work we present the idea of adding UTASA to Uppaal, such that slicing occurs transparently to the user and the idea of graphically show the slices compared to the original model is also presented as future work.

5.2 The UTAP library

The UTAP library is the UPPAAL Timed Automata Parser. UTAP has the ability to parse and type check UPPAAL models in any of the three file formats supported by UPPAAL. Table 5.1 gives the core classes and their use.

TimedAutomataSystem	The abstract syntax tree (AST) created by the library. The class is implemented using the visitor pattern, allowing it to be visited.
ParserBuilder	Represents the interface used by the parser to build the AST. The primary implementor of the interface is the *SystemBuilder*.
SystemBuilder	The most specialized class in the builder hierarchy (see figure 5.2(a)) also implementing the *ParserBuilder* interface. The class is used by the parser (implemented using bison) to build the (AST)
StatementBuilder	Used in the builder hierarchy to construct Statements. *StatementBuilder* is a sub-class of *ExpressionBuilder*.
ExpressionBuilder	Used in the builder hierarchy to construct expressions.
SystemVisitor	The most specialized class in the visitor hierarchy. Used to visit an instance of *TimedAutomataSystem*. The class is designed to visit the automata related components of the AST, inheriting functionality to visit imperative code from *StatementVisitor*.
StatementVisitor	Used to visit statements in an instance of *TimedAutomataSystem*.
ExpressionVisitor	Used to visit expressions in an instance of *TimedAutomataSystem*.

Table 5.1: Summary of the main classes of UTAP.

There are multiple ways to use the library. The simplest approach, which is also the one used in our implementation, parsing is initiated by calling the function: (defined in "utap/utap.h")

parseXTA(file, system, false)

In the call to parseXTA, the first argument is the XML file to read (input). The second is (output) a reference to a *TimedAutomataSystem* object (initially empty), which is to be buildt by the parser. The third is a flag indicating whether to use the new or the old UPPAAL syntax.[1]

[1]The old syntax is the one used in UPPAAL 3.4, the new is the one used in UPPAAL 3.6 and later versions.

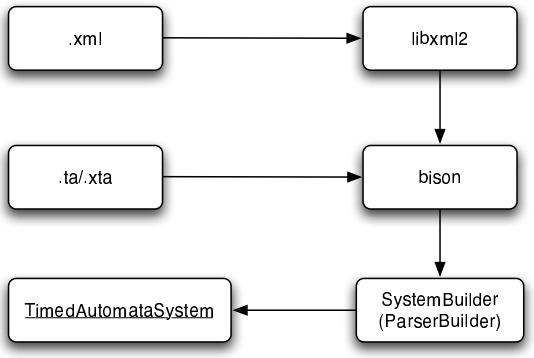


Figure 5.1: Initial information flow through the UTAP library.

5.2.1 Architecture And Usage

Figure 5.1 shows the initial information flow through the UTAP library.

As illustrated in Figure 5.1 the parser, which is implemented using the bison framework, is capable of parsing both old and new UPPAAL syntax. It therefore comes with functionallity to read .ta and .xta files directly. To handle XML files, the parser uses the libxml2 library[2] to do the initial parsing and the result is then "fed" to the bison parser.

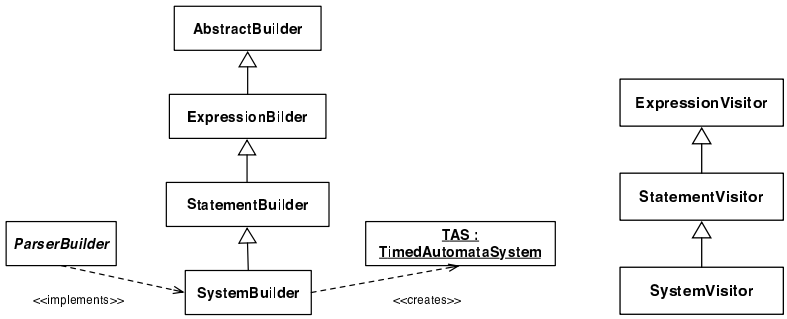
During the parsing of tokens, the parser uses the abstract interface provided by the *ParserBuilder* class in order to build the abstract syntax tree, which is an instance of the *TimedAutomataSystem* (TAS) class. The methods in the *ParserBuilder* class are implemented by the *SystemBuilder* class, which is a sub-class of *Expression-Builder*, which is then again a sub-class of *AbstractBuilder* see Figure 5.2(a). The *SystemBuilder* writes the model to an instance of the *TimedAutomataSystem* (TAS) class.

The design abstracts the difference between the .XML, .xta and .ta input formats and also hides the differences between the 3.4 and 3.6/4.0 formats from any implementation of the *ParserBuilder* interface.[3]

A TAS object represents the templates, variables, locations, edges and processes of a model. Symbols are represented by *symbol_t* objects. Symbols represent unique identifiers, such as names of variables and functions. Expressions and variables in the AST are decorated with types, which are represented by a *type_t* object. Symbols are grouped into frames (represented by *frame_t* objects). Frames are used to represent scopes and other collections of symbols such as records or parameters of templates and functions.

As usual, expressions are represented using a tree structure, explicitly representing

[2]See http://xmlsoft.org
[3]For equivalent input, the parser will call the same methods in the *ParserBuilder* class.



(a) The builder hierarchy.

(b) The visitor hierarchy.

Figure 5.2: Overview of UTAP.

precedence, where the leaves represent constant values or symbols and the inner nodes represent operations. Each node is referenced using an *expression_t* object.

Finally, in order to traverse the generated TAS, a visitor interface (visitor pattern [44]) is provided by UTAP. See Figure 5.2(b) for an overview.

5.3 The UTASA Library

The UTASA library is used to slice a TAS buildt by UTAP. In the following we show the overall architecture of the library and provide a description of the main data structure. Furhtermore, we discuss current issues and possible improvements.

5.3.1 Architecture

The core of the UTASA architecture is based on two abstraction levels. An intermediate abstraction over the basic edge and state types used in UTAP, which provides a connected graph over the automata, used to collect and update dependencies relative to updates, guards and invariants. The later, second, abstraction provides the SDG which is used by the algorithm to obtain the slicing set. Figure 5.2 presents an overview of the classes composing the UTASA library.

5.3.2 Modifying The AST

Given the fact that the AST *TimedAutomataSystem* provided by UTAP is not designed to accept changes, UTASA initially employes the help of two classes designed to copy the AST, while accepting changes. Figure 5.3 shows both.

Before the SDG can be build (and the slicing algorithm run) UTASA uses the *NormalFormCopyer* class, a sub-class of *SystemCopyer*, to copy the input system to a temporary form, in which the imperative components are represented using a subset

SdgBuilder	Responsible for creating the full SDG structure and hereby collecting the dependencies collected by all instances of *InstanceSdgBuilder*.
InstanceSdgBuilder	Responsible for collecting all the dependencies computed by *AutomataSdgBuilder* and *FdgBuilder*.
AutomataSdgBuilder	Maintains data and control dependencies at automaton-level.
FdgBuilder	Maintains intra and inter-functional dependencies within an automaton.
SystemCopyer	Used to simply copy a TAS. Also used to copy a TAS with respect to a given slicing set
NormalFormCopyer	Used to convert the input system into a subset of SSA form.
DagBuilder	Used to build a DAG representation of the input automaton.
Location	An abstraction for the state_t used in UTAP and used in the DAG.
Edge	An abstraction for the edge_t used in UTAP and used in the DAG.

Table 5.2: Summary of the main classes of UTASA.

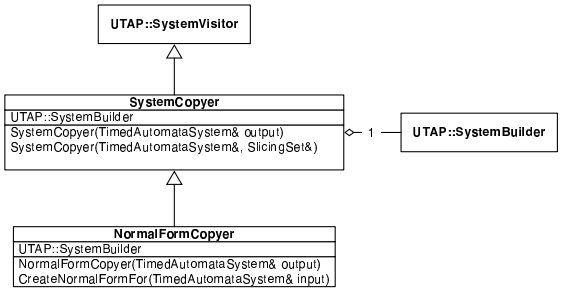


Figure 5.3: Classes used to copy *TimedAutomataSystems*.

of the SSA form. This form is exploited by the later SDG construction, since the control-flow of the system is now much more predictable.

The *SystemCopyer* is used for two purposes; simply copying a TAS and copying a TAS with respect to a slicing criteria. After running the slicing algorithm, which produces a slicing set for the system on normal form, *SystemCopyer* is used for copying the temporary TAS out of normal form, while also slicing the TAS. In order to construct the slice, an extra constructor is needed for *SystemCopyer*, one that also takes a slicing set. This is used to check whether or not a construct in the original model should be included in the output system.

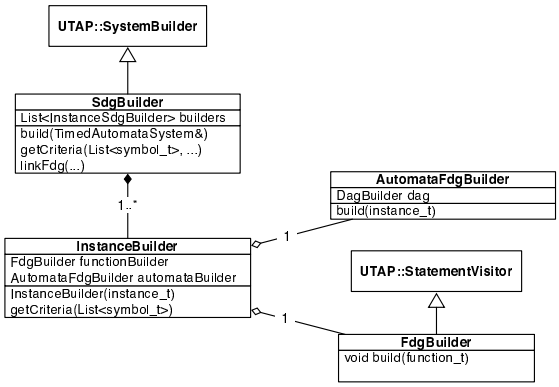


Figure 5.4: Classes used to build the SDG structure

5.3.3 SDG Construction

Construction of the SDG data structure (see page 76) is done using a hierarchy of builders, most of which are implemented using the visitor interface provided by UTAP. The top-level builder used to create the SDG, is the *SdgBuilder*. The *SdgBuilder* is comprised from one or more *InstanceBuilders*, depending on the number of timed automata in the model, each of which consist of one *AutomataFdgBuilder* and one *FdgBuilder*. Figure 5.4 illustrates the classes and the hierarchy.

The primary responsibility of the *FdgBuilder* is to collect all possible intra-functional dependecies. That is, control and data dependencies within the body of a function. Furthermore, it has the extra responsibility of establishing function linkage, hereby creating inter-functional dependencies.

The *AutomataFdgBuilder* has the primary responsibility to collect all possible dependencies at the automaton-level. That is, it makes sure that all possible paths in the automaton (loops included) are explored and propagates dependencies to updates, guards and invariants. Like the *FdgBuilder*, *AutomataFdgBuilder* also has the responsibility to establish function linkage.

InstanceSdgBuilder is the class that combines the dependencies from the *FdgBuilder* and the *AutomataFdgBuilder*. It makes sure that each function of the automaton is passed to the functionBuilder (an instance of *FdgBuilder*) and that the automaton is passed to the build function of automataBuilder (an instance of *AutomataFdgBuilder*). The function *getCriteria(list<symbol_t>)* is used to collect the slicing criteria for that specific automaton instance using the provided list of symbols.

Finally, the *SdgBuilder* class is responsible for combining the dependencies computed for the entire system. That is, the *SdgBuilder* is responsible for linking all the FDG dependencies, hereby constructing the SDG, see 5.3.5 for further details about the SDG data structure. The *getCriteria(list<symbol_t>, ...)* function is used to collect the overall slicing criteria of the system and is therefore the "entry-point" to the SDG for the slicing algorithm.

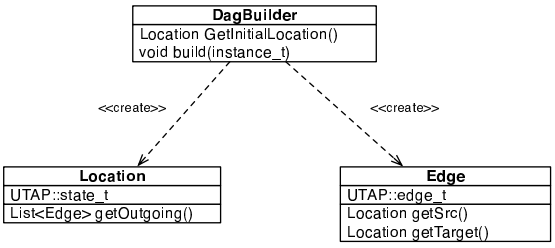


Figure 5.5: Classes used to build the DAG structure.

5.3.4 Dependency Analysis of An Automaton

Figure 5.5 shows the classes used to build a data structure representing timed automata as directed acyclic graphs (DAG). The class *DagBuilder* is the main class of the structure and is comprised of *Location* objects and *Edge* objects. The main responsibility of *DagBuilder* is, of course, to build the DAG, but as a secondary task, it must also provide the initial location, used in the *AutomataFdgBuilder*, described above. The classes *Location* and *Edge* is used to abstract the constructs *state_t* and *edge_t* from UTAP, in order to ease the computation of the DAG.

Computing Reaching Information

The data-flow information required to construct the SDG is computed using reaching information, as described in Section 4.2.2. Reaching may be determined somewhat trivially for imperative languages, as the control-flow of a C-like language only have loops and branching (if-statements). In contrast, an automata may have a much more complex control-flow, where loops may be entered at multiple points.

The computation of reaching information for timed automata is therefore based on a fix-point strategy. Starting at the initial location, each program point is decorated with reaching information, which is then propagated along the control-flow. All the edges in the automaton are visited and all updates are recorded for all possible control-flows, until no change in the reaching information occur. That is, the decorations have converged.

5.3.5 The SDG Data Structure

In order to implement the SDG described in Section 4.2.3, a specialized data structure has been developed. The data structure maintains all possible dependencies in the TAS and provides a means for the slicing algorithm to explore these, in order to compute the slicing set.

In order to obtain the SDG structure, we create an *SdgNode* (object) for every vertex in the control-flow graph, as described in Section 4.2.3. The graph is as expected composed of *SdgNodes*, holding references to other *SdgNodes*. That is, the graph is maintained by mutual references between nodes.

Since the algorithm is based on backwards traversal of data-flow- and control-flow-edges, the design of the SDG is based on inverted edges, refered to as data dependency and control dependency edges. Each dependency is implemented as a reference from an *SdgNode* to the *SdgNode* on which it is dependent. Each node in the graph contains a list of references representing data dependencies to other nodes. Likewise, it contains a reference to a single node, representing control dependency.

5.3.6 Issues

During implementation, we decided not to do the implementation with respect to structs and call-by-reference parameters. This was done solely for the reason that we only wanted to make a proof-of-concept implementation of the theory described and as these construct do not add much to the complexity, they would only add to the lines of code produced. Furthermore, as the implementation is developed for the purpose of producing a prototype for the theory, efficiency has not been our main concern.

Moreover, we do not support the template feature of Uppaal because of the fact that UTAP does not provide the instantiated models. If UTAP were able to provide this, UTASA would of course support this feature. The reason for the restriction to instantiated models is that we expect better slicing result, compared to just slicing the templates. Furhermore, the sink location described in Section 4.3 is also not implemented. We leave this for future work.

Chapter

6

Experimental Results

This Chapter describes the experiments done using our slicing implementation on two Uppaal models. The first model, the Train-Gate example, is extended with extra constructs and the second model, the "real life" example, is sliced unmodified. We state a common set of metrics and conduct the measurements both before and after slicing has been done.

The tests performed substantiate the slicing technique presented in Section 4. Furthermore, it demonstrates that the implementation presented in Chapter 5 is able to slice the presented Uppaal models and that it is able to improve the performance drastically.

We finish the Chapter with a summary and discuss the results obtained from the experiments.

Contents

6.1 Introduction

The experiments conducted in this Chapter show the strenghts of slicing for UPPAAL. The experiments presented are based on two case models:

Train-Gate: As explained in Section 1.2.1, the Train-Gate example, which we reuse here, has been modified to make the example interesting for the purpose of slicing. We have added extra statements used only for visual aid for the user. Meaning, the extra statements do not add to the functionality of the model, but only act as heuristics to the user, in case of e.g. debugging.

Mapper: In addition to the Train-Gate example, we have included a "real life" example possessing similar design problems. It shows how a group of students have used UPPAAL to model a small part of their system and how the model itself has a couple of heuristic variables.

For a quick overview of the test results for both models, see Tables 6.1 and 6.2.

6.2 Test Metrics

Both models are tested based on the same set of metrics. The metrics used in the experiments are:

- VT = Verification Time.
- MU = Memory Usage.
- SS = Symbolic States.
- NS = Number of Statements.
- NV = Number of Variables.

Both the original and sliced version of the models are monitored during verification (of the same property) and the maximal memory consumption is recorded along with the number of symbolic states explored. The experiments are conducted using the command-line tool *verifyta*, which comes with the standard UPPAAL distribution, to perform the verification and monitor the symbolic states explored. In order to record the memory consumption, we use the tool memtime[1] which is the default performance (memory) measuring tool used by the UPPAAL development team. Finally, we manually count the number of statements and variables in the model, in order to show the syntactic reduction performed by slicing.

6.2.1 Test Environment

The experiments are conducted on a medium range workstation, in order to obtain a realistic impression of the optimizations. The test machine is a 2.6Ghz P4 (w.o. HT) with 512Mb memory and a standard ATA disk system. Since these specifications might be considered low standard by some, we have re-run some of the verification

[1] See http://freshmeat.net/projects/memtime/ for further information about this tool.

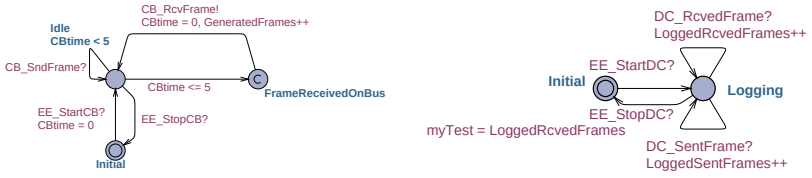
tasks, which ran out of memory, on a quad-core operton SUN server with 4GB of memory (also the maximal amount of memory that UPPAAL can allocate, since it is 32 bit). None of the verification jobs, which was re-run, exhibited any improvement on the server hardware i.e. 4GB memory was not enough to redeem the situation.

6.3 Real Life Example - Mapper

The real life example we have chosen is provided by students at the department of communication technology at Aalborg University. The intention of the model, which is a network of the automata shown in Figures 6.1, 6.2, A.1, A.2 and A.3, is to verify a part of a test tool used for analysing the traffic on a Controller Area Network (CAN). Only Figures 6.1 and 6.2 are shown in this Section because these are the ones involved in the slicing. Moreover, the tool should also be able to generate traffic by itself. The tool implements a traffic generator, which is able to run on any network. The part of the implementation responsible for translating from the tool's architecture to the test invironment's network protocol, and vice versa, was named the mapper. The mapper contains the following global declarations:

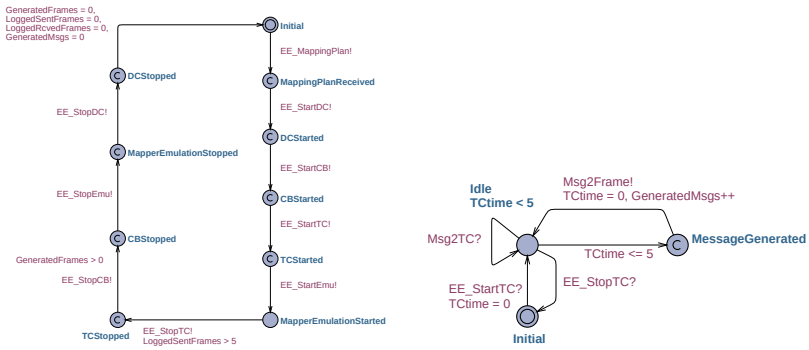
```
chan PlanReceived, CloseDriver, OpenDriver, StartEmu, StopEmu,
EE_MappingPlan, EE_PlanReceived, EE_StartEmu, EE_StopEmu,
EE_StopDC, EE_StopTC, EE_StartTC, EE_StartDC, Frame2CB,
Frame2Msg, Msg2TC, Msg2Frame, DC_SentFrame, DC_RcvedFrame,
SentFrame, RcvedFrame, CB_SndFrame, CB_RcvFrame, EE_StartCB,
EE_StopCB;

clock CBtime, TCtime;

int GeneratedMsgs = 0, GeneratedFrames = 0, LoggedRcvedFrames = 0,
LoggedSentFrames = 0;
```

As seen in the model, there are three stubs and one driver. The three stubs each consist of a data collector, which is responsible for logging traffic. The traffic controller is responsible for generating traffic and the bus is both able to send and recieve frames. The driver both activates and de-activates the process. Furthermore, it models a users input to the system. The mapper model is only part of the full system and it was the only part chosen, by the students, to model in UPPAAL.



(a) CBstub (b) DCstub

Figure 6.1: Parts of the Mapper model.



(a) EEdriver (b) TCstub

Figure 6.2: Parts of the Mapper model.

6.3.1 Test Results

In the case of the mapper model, we chose a simple but vital property for testing. The property used, expressed that the system under all circumstances was to be *deadlock* free (`A[]not deadlock`). Running the verification (before slicing) on the test machine, immediately consumed all resources and verification continued 2 hours without result.

Running the *TASlicer* application on the mapper removed the three unbounded integer variables `GeneratedMsgs`, `GeneratedFrames` and `LoggedRcvedFrames`. The only purpose of the integers was to model statistics of the protocol. Moreover, the variables were never referenced for any other purpose than to update their values (see Figure 6.1(b) and 6.2(b)). The fourth (remaining) variable is preserved since it is used in a guard of the EEdriver (Figure 6.2(a)).

Re-running the verification job with the sliced model, *verifyta* exited after 11.12 seconds, reporting an error; indicating a flaw in the model design (a flaw which otherwise might have gone undetected). After the model had been reviewed by members of the Uppaal team, the experiment was redone.

At this point the model had been optimized to such an extent, that slicing could not provide a noticeable difference in performance gain, but the syntactic reduction remained, i.e. the reduction in the number of variables and statements. Furthermore, the number of symbolic states were reduced drastically. From 2074 to 199. The results are summarized in Table 6.1.

6.4 The Extended Train-Gate Example

The Extended Train-Gate Example is modelled as described in Section 1.2.1, but with only 4 trains and the test metrics are as described in Section 6.2.

	Mapper Example				
Deadlock free	VT	MU	SS	NS	NV
Before Slicing	N/A	4GB+	85587630+	12	6
After Slicing	11.12sec ⋆	66572KB	786391	6	3
After Fix	VT	MU	SS	NS	NV
Before Slicing	0.11sec	2862KB	2074	12	6
After Slicing	0.10sec	2852KB	199	6	3

Table 6.1: The test results of mapper experiments. ⋆verification failed

6.4.1 Test Results

The experiments conducted using the train-gate model, are based on two properties, as with the mapper model, we check for the absence of deadlocks (`A[]not deadlock`). Furthermore, we verify for each train, the possibility of crossing (e.g. `E<> Train1.Cross`).

Running the verification job on the un-sliced train-gate model, deadlock freeness reacted similarly to the mapper model; the verification process was allowed to run for two hours before it was terminated manually. After running the *TASlicer* application on the model, verification was drastically improved, finishing in 0.2 seconds, confirming the model to be deadlock free. Alternatively, the second property did, although showing a syntactic reduction, not exhibit any improvement. The results are summarized in Table 6.2.

	Train-Gate Example				
Deadlock free	VT	MU	SS	NS	NV
Before Slicing	N/A	4GB+	77636326+	34	14
After Slicing	0.2sec	2848KB	413	28	10
Train may cross	VT	MU	SS	NS	NV
Before Slicing	0.11sec	2856KB	14	34	14
After Slicing	0.10sec	2848KB	14	28	10

Table 6.2: The test results of the Train-Gate experiments.

6.5 Summary

As expected, the syntactic reduction had a noticeable impact on verification. The fact that the slicer removed four integer variables in the train-gate example, would theoretically reduce the state-space by 4×2^{16} states. That is, the reduced state-space is $\frac{[state\text{-}space]}{4 \times 2^{16}}$. Although this is a considerable reduction, the growth in the value of integer variables does not affect the fact that each train may reach a state where they have been allowed to cross. Similarly, in the mapper case, the results show that verification of the unmodified model does benefit greatly from slicing, but a manually optimized model may not require slicing to obtain an acceptable verification time.

In order to substantiate our claim that software engineers tend to decorate their models, we leave for future work to perform further experiments on more UPPAAL timed automata models.

Chapter

7 Conclusion and Final Remarks

In this Chapter, we present future work and other related issues. Furthermore, we conclude on the project and give our final remarks.

Contents

7.1 Conclusion

The goal of this thesis has been to research program slicing based on static analysis in order to obtain a syntactic reduction of UPPAAL models. It was our initial expectation that slicing would be a substantial contribution to the UPPAAL tool, since a syntactic reduction would directly influence the size of the concrete state-space. Although UPPAAL already employes sophisticated techniques to reduce the required exploration (see Chapter 3), we have been able to show that slicing does indeed increase the performance of the tool and in some cases, it may even enable verification of properties, which would otherwise fail to complete.

Inspired by several articles and related work, we have shown how to create reachability preserving slices of models, expressed using the extended timed automata formalism (see Section 2.3). Primarily driven by the fact that the extended timed automata formalism (as well as the UPPAAL language) allows the definition and use of auxiliary functions, we have initially explored existing work focusing on inter-procedural analysis. Based on our research, we have chosen to base our approach on a notion of *system dependency graphs* (see Section 4.2.3). Although we expect that the SDG structure was originally conceived for the purpose of representing imperative code, we show that it may also be used to represent the dependencies imposed by the structure of timed automata. Using the SDG as a common representation of dependencies creates a uniform abstraction over the hybrid control-flow, generated by the combination of timed automata and imperative code. Furthermore, the SDG is highly suitable for analysis using graph theory, which in-turn may help produce a more precise slice of the model.

In addition to proving the correctness of the approach (Chapter 4) in terms of the extended timed automata formalism, we have presented a prototype implementation (UTASA - Chapter 5), extended to the language of UPPAAL. Although the experiments, which we have documented in (Chapter 6), show that slicing will assist the user in removing irrelevant "satelite" data and in turn reduce the time and resources required to verify properties of models, the approach which we propose is, atleast from a theoretical perspective, also able to reduce more substantial parts of models. Eventhough we have not conducted experiments in this area, the use of sink locations (Section 4.3) may be used to remove parts of a model, which cannot affect the outcome of the verification. A reduction in the structure of a model will most likely entail even greater reductions of the state-space, yielding a shorter verification time.

Theoretical Results

In Chapter 4 we introduce an approach for reachability preserving slicing of models expressed in the extended timed automata formalism presented in Chapter 2. Furthermore, Section 4.6 in Chapter 4 contains formal proofs, showing that the $E\Diamond$-Bisimulation is in-fact reachability preserving (Theorem 19). The result of Theorem 19 is then used to prove correctness of the slicing approach, by proving that the relation $\simeq$, which relates a model and its slice, is indeed a $E\Diamond$-Bisimulation (Lemma 21).

Implementation

A prototype implementation has been developed in order to show that the slicing approach, described in Chapter 4, can in-fact be extended to the language of Uppaal. Furthermore, the implementation also shows that static analysis can be very beneficial to verification tools like Uppaal.

In Chapter 5 we document the UTAS library and we give a small overview of the architecture of UTAP (The Uppaal Timed Automata Parser) used by UTASA. UTASA is developed as a preprocessing library for Uppaal. That is, slicing is available before verification. Furthermore, as UTASA contains several data structures designed for static analysis, we encourage the further development of UTASA.

Results of The Experiments

As seen in Table 7.1, UTAS is able to slice away irrelevant variables, hereby reducing the number of symbolic states explored significantly, which is also the reason for the much lower verification time and memory usage.

The acronyms are as follows:

- VT = Verification Time
- MU = Memory Usage
- SS = Symbolic States
- NS = Number of Statements
- NV = Number of Variables

	Train-Gate Example				
Deadlock free	VT	MU	SS	NS	NV
Before Slicing	N/A	4GB+	77636326+	34	14
After Slicing	0.2sec	2848KB	413	28	10
Train may cross	VT	MU	SS	NS	NV
Before Slicing	0.11sec	2856KB	14	34	14
After Slicing	0.10sec	2848KB	14	28	10
	Mapper Example				
Deadlock free	VT	MU	SS	NS	NV
Before Slicing	N/A	4GB+	85587630+	12	6
After Slicing	11.12sec *	66572KB	786391	6	3
After Fix	VT	MU	SS	NS	NV
Before Slicing	0.11sec	2862KB	2074	12	6
After Slicing	0.10sec	2852KB	199	6	3

Table 7.1: Summary of the test results made on the Extended Train-Gate Example and the Mapper Example.

As Table 7.1 shows, UTASA works for Uppaal timed automata and is clearly beneficial.

7.2 Future Work

In the following, we present future work, which we believe would be extremely beneficial or crucial for the use of slicing with Uppaal.

7.2.1 Structs and Call-by-references Parameters

In the prototype implementation, we currently do not support structs and call-by-reference parameters. Therefore, it is necessary to implement functionality to handle these constructs, in order to support the full language of Uppaal.

7.2.2 Implementing Algorithm 2

The current implementation is based on Algorithm 1 and therefore suffers the calling-context-problem described in Section 4.1.1. An implementation of Algorithm 2 would solve this problem, hereby providing a more precise slice of the system. The models we have tested will not show a difference using Algorithm 2, but in the general case, it generates a more precise slice of the system.

7.2.3 UTASA as Part of Uppaal

The UTASA prototype library is a preprocessing tool for Uppaal. It is not yet an integrated part of Uppaal and must therefore be used manually by potential users. It would be beneficial to incorporate it into Uppaal, such that slicing becomes transparent to the user (alternatively graphical visualization could be employed).

7.2.4 Slicing The Structure of Timed Automata

In the current work, we present a slicing approach which does not attempt to reduce the structure of the automata in the model. Using theory presented in [37] it would be possible to create a CTL preserving reduction of the model, in-turn reducing the state-space even further. Although we already introduced the possiblity of having a sink location, we do not discuss how to compute irrelevant locations.

7.2.5 Further Experiments

We claim that software developers tend to decorate their models with auxiliary data and in order to substantiate this, further experiments could be performed on newly collected models from industry software developers.

7.2.6 Code-Review and Improvements

As ATASA is a prototype implementation it would be very beneficial to get a skilled C programmer to review the code. Our main effort has been the functionality of the code and not efficiency. A code review could most likely improve the performance of the library and also help remove potential bugs.

7.3 Related Future Work

In this Section we give some ideas to peripherical future work and our final remarks.

7.3.1 Over-approximate and Refine

In order to further extend UTASA, the approach in [26] could be applied. In [26], they automatically over-approximate the given model and since it is an over-approximation, the absence of an abstract counter-example implies the absence of counter-examples in the full model. They automate all the steps of the abstract-refinement loop (known also from lazy abstraction used in BLAST [32]), in the setting of real-time specifications that are given in terms of PLC-Automata. The abstraction starts with the coarsest possible abstraction and iterates as long as spurious counter-examples are found. If the model-checker finds an abstract counter-example, then a counter-example analyser is used to check whether it is spurious or not. They argue that in most cases, an over-approximation is sufficient to establish whether or not a certain property holds for the given system.

7.3.2 Visualizing Program Slices

Thus far, the emphasis of most slicing research has been on algorithmic aspects. Little attention has been paid to the question of how slices could best be visualized and interactively displayed or browsed. In [5] they present a technique for visualizing program slices and also present a tool called SeeSlice to demonstrate the visualization. Future work could include applying this technique for Uppaal, building on top of the previous mentioned slicing techniques.

7.3.3 Dynamic Slicing in Uppaal

The approach taken in our work is that of static slicing, but we suspect that dynamic slicing could, as done in [15], add an even greater reduction in the state-space, as we most likely would be able to slice away whole automata from the system dynamically. We expect this to be beneficial in the general case and it would of course be necessary to apply some changes in the Uppaal engine to support this dynamic slicing. Furthermore, a comparison of dynamic and static slicing of Uppaal models would be interesting in order to evaluate the true benefits of dynamic slicing.

7.3.4 Templates of Uppaal

At the moment, UTASA does not support the template feature of Uppaal due to the fact that UTAP does not provide the instantiated timed automata models. If a future version of UTAP is able to provide this, then UTASA would be able to support the template feature by default.

Appendix

A Appendix

A.1 The Mapper Model

For completeness we provide the remainder of the Mapper model.

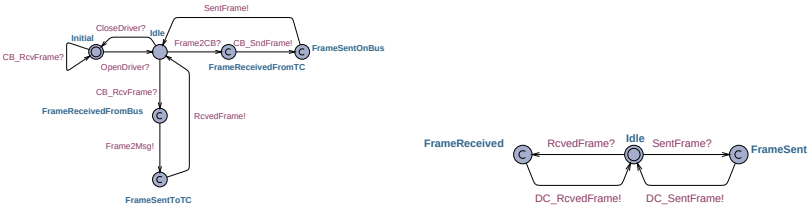


Figure A.1: Parts of the Mapper model.

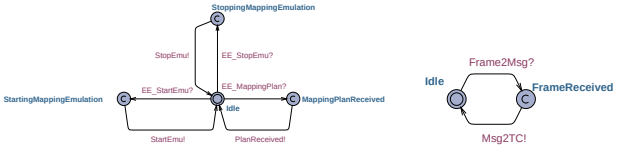


Figure A.2: Parts of the Mapper model.

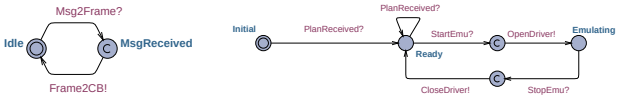


Figure A.3: Parts of the Mapper model.

References

[1] Alur, Courcoubetis, and Dill. Model-checking for real-time systems. In *LICS: IEEE Symposium on Logic in Computer Science*, 1990.

[2] Alur, Courcoubetis, and Dill. Model-checking in dense real-time. *INFCTRL: Information and Computation (formerly Information and Control)*, 104, 1993.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[4] Thomas Ball. The SLAM project: Debugging system software via static analysis. pages 1–3.

[5] Thomas Ball and Stephen G. Eick. Visualizing program slices. In *Visual Languages*, pages 288–295, 1994.

[6] Thomas Ball and Sriram K. Rajamani. The SLAM toolkit. *Lecture Notes in Computer Science*, 2102:260–??, 2001.

[7] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL implementation secrets. *Lecture Notes in Computer Science*, 2469:3–??, 2002.

[8] Gerd Behrmann, Patricia Bouyer, Emmanuel Fleury, and Kim G. Larsen. Static guard analysis in timed automata verification. In Hubert Garavel and John Hatcliff, editors, *Proceedings of the 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 254–277, Warsaw, Poland, April 2003. Springer.

[9] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[10] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[11] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[12] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.

[13] Bouyer. Untameable timed automata! In *STACS: Annual Symposium on Theoretical Aspects of Computer Science*, 2003.

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

[15] Víctor A. Braberman, Diego Garbervetsky, and Alfredo Olivero. Improving the verification of timed systems using influence information. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2002.

[16] J. Büchi. Weak second-order logic and finite automata. *Z. Math. Logik Grundlagen Math.*, 5:66–92, 1960.

[17] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science (TCS)*, 240(1):177–213, 2000.

[18] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–265, April 1986.

[19] 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 Transactions on Programming Languages and Systems*, 13(4), October 1991.

[20] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer-Verlag, 1996.

[21] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. pages 313–329.

[22] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212, Berlin, June 1990. Springer.

[23] E.A. Emerson and E.M. Clarke. Using Branching-Time Temporal Logic to Synthesize Synchronization Skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[24] Michael D. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Redmond, WA, USA, 26 July 1994.

[25] Marius Bozga et al. Tools and applications ii: The if toolset. volume 3185. Springer, 2004.

[26] No Author Given. Automatic abstraction refinement for timed automata.

[27] Hatcliff and Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In *CONCUR: 12th International Conference on Concurrency Theory*. LNCS, Springer-Verlag, 2001.

[28] John Hatcliff, James Corbett, Matthew Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1999.

[29] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder, September 12 1998.

[30] Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Inf. Comput*, 117(2):221–239, March 1995.

[31] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[32] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. pages 58–70.

[33] C. A. R. Hoare. Communicating sequential processes. *Comm.A.C.M.*, 21(8):666–677, August 1978.

[34] Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003.

[35] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM Press.

[36] Radu Iosif and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *Proc. of the 6th International SPIN Workshop*, volume 1680 of *LNCS*, pages 261–276. Springer-Verlag, September 1999.

[37] Agata Janowska and Pawel Janowski. Slicing of timed automata with discrete data. pages 181–195. Fundamenta Informaticae, 2006.

[38] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[39] K.L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

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

[41] Kim G. Larsen and Jiri Srba. *Semantics and Verification*. AAU, 2006.

[42] M. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of Banff'94*, 1994.

[43] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[44] Gamma. Helm. Johnson og Vlissides. *Design Patterns, Elements of reusable Object-Oriented Software*. Addison-Wesley profeccional computing series. Addison-Wesley Publishing Company, 1995.

[45] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[46] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, 1995.

[47] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[48] Cormen. Leiserson. Riveset and Stein. *Introductions to Algorithms*. the MIT press, 2 edition, 2002.

[49] Davide Sangiorgi and David Walker. *The Pi-Calculus — A Theory of Mobile Processes*. Cambridge University Press, 2001.

[50] Michael I. Schwartzbach. Lecture notes on static analysis.

[51] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang*, 3(3), 1995.

[52] Mark D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.