CARL VON OSSIETZKY universität OLDENBURG
DEPARTMENT OF COMPUTING SCIENCE
CORRECT SYSTEM DESIGN GROUP

Diploma thesis

SAT-based Verification for Abstraction Refinement

Stephanie Kemper

January 4th, 2006

Responsible Supervisor: Prof. Dr. Ernst-Rüdiger Olderog
Second Supervisor: Prof. Dr. Martin Fränzle
Advisor: Dipl.-Inform. André Platzer

Abstract

The aim of this diploma thesis is to verify reachability properties of timed automata using SAT-based verification methods, while mainly eliminating the *state explosion* problem using *abstraction refinement* techniques. Especially for complex and safety critical systems, reachability analysis plays a major role, thus, verification shall be restricted to the system parts essential to the property.

This thesis will present an iterative approach to automatic verification of reachability properties. While the essential, non-abstractable system parts have to be "guessed" during the first cycle, from begin of the second it is possible to acces former results and identify wrongly omitted system parts to include them again.

The timed automaton and the property are firstly translated into formulae, such that abstraction reduces to modification of formulae and these may be verified using a SAT-based model checker. Identification of wrongly omitted system parts and subsequent refinement are realised using craig interpolants resulting from false negatives.

The efficiency and universality of this approach is shown using several examples.

Zusammenfassung

Das Ziel der vorliegenden Diplomarbeit ist es, Erreichbarkeitsfragen von Realzeitautomaten mittels SAT-basierter Methoden zu verifizieren, wobei das *state explosion* Problem mit Hilfe der *Abstraction-Refinement* Technik weitestgehend eliminiert wird. Erreichbarkeitsfragen spielen besonders bei komplexen sicherheitskritischen Systemen eine Rolle, so dass die Überprüfung dieser Eigenschaften auf die für sie wichtigen Systemteile eingeschränkt werden soll.

Es wird ein iteratives Verfahren vorgestellt, welches die Verifikation von Erreichbarkeitsfragen automatisiert. Müssen dabei im ersten Anlauf die für die Eigenschaft wichtigen, nicht abstrahierbaren Teile noch "geraten" werden, so ist es ab dem zweiten Durchlauf möglich, auf vorherige Ergebnisse zurückzugreifen und Systemteile, welche fälschlich weggelassen wurden, zu identifizieren und wieder in die Berechnung mit einzubeziehen.

Konkret werden Automat und die Eigenschaft zunächst in Formeln übersetzt, so dass Abstraktion sich auf die Modifikation von Formeln reduziert und diese dann mit Hilfe SAT-basierter Modelchecker überprüft werden können. Die Identifikation fälschlich weggelassener Systemteile und das anschließende Refinement wird mit Hilfe von Craig Interpolanten ermöglicht, welche aus falschen Gegenbeispielen resultieren.

Die Effizient und Allgemeingültigkeit des Ansatzes wird anhand von Beispielen verdeutlicht.

Danksagung

Ich möchte all jenen danken, die mir während der Zeit der Erstellung dieser Diplomarbeit mit Rat und Tat, aber auch mit Freundschaft und Nachsicht zur Seite gestanden haben.

Zunächst möchte ich mich bei Prof. Dr. Ernst-Rüdiger Olderog und Prof. Dr. Martin Fränzle bedanken für die Möglichkeit, eine Diplomarbeit über ein solch interessantes Thema zu schreiben. Herr Prof. Olderog weckte bereits durch zahlreiche Vorlesungen mein Interesse an Model Checking und Realzeitsystemen.

Ein besondere Dank gilt meinem Betreuer Dipl.-Inform. André Platzer für die wirklich gute Betreuung: Er hat mit mir in vielen Diskussionen die verschiedenen Aspekte meiner Arbeit beleuchtet, wobei mir stets genügend Freiraum für eigene Ideen und Ansätze blieb. Wenn ich doch einmal nicht weiterwusste, so hatte André stets die richtige Frage parat, durch deren Beantwortung ich mir selber weiterhelfen konnte. Seine motivierende Art hat mir über die schwierigen Phasen der Diplomarbeit hinweggeholfen. Auch waren Andrés Korrekturen der vorliegenden Arbeit für mich von großem Wert.

Ein großer Dank geht auch an Andreas Schäfer, Jochen Hoenicke und Michael Möller, die immer Zeit für meine Fragen hatten. Für Implementierungsfragen hatte insbesondere Michael stets ein offenes Ohr.

Alexander Borgerding und Rico Starke gebührt mein Dank dafür, dass sie diese Arbeit Korrektur gelesen haben.

Bei Ken McMillan möchte ich mich bedanken für die Erklärungen und Hilfen zu dem von ihm entwickelten Tool FOCI.

Meiner Familie gebührt Dank für die Unterstützung während des ganzen Studiums.

Ganz besonders herzlich möchte ich mich bei meinem Ehemann Nils bedanken. Für ihn war das letzte halbe Jahr mit Sicherheit genau so schwer wie für mich, und trotzdem hat er sich nie etwas anmerken lassen, wenn ich bis spät in die Nacht gearbeitet habe. Er hat mich im Gegenteil nach Kräften unterstützt und immer Verständniss gezeigt. Ohne ihn wäre diese Arbeit nicht möglich gewesen!

Contents

List of Figures

1. Introduction

1.1. Model Checking

To an increasing degree, software and hardware as well as algorithms and procedures nowadays are expected to be fault-tolerant and robust. For this reason, systems need to be simulated in order to guarantee desired behaviour, ensure that specified properties hold and prevent erroneous execution. If simulation has been successful and errors have been detected, expensive production costs and costly amendments may be avoided.

Model checking is one of the major techniques to verify properties of finite state systems. It basically consists of three steps, see also (Clarke *et al.*, 1999):

1. Modelling: The system description (mainly in natural language) is translated to some formal description. The formal description is used as input for the model checking tool and thus depends on the accepted input language. Common representations are formulae, Kripke structures and timed automata (which will be used within this thesis). Surely depending on the level of abstraction already used when describing the system, the modelling task can often be done automatically.

2. Specification: A description of the desired behaviour of the system and its properties it has to fulfil is given. This is usually done by means of temporal logic, as this allows to specify behaviour of the system over time. Attention has to be paid to ensure no unwanted side effects appear (due to too few constraints).

3. Verification: The formal descriptions of the system and the properties are checked to be consistent. This is mostly done automatically, only the results have to be analysed and interpreted manually.

Depending on the results of the verification step, the natural language system description or formal system description are adapted (erroneous initial description in the former case and erroneous system description or translation in the latter one), or some properties are added to the formal specification (incomplete specification).

With an appropriate translation of properties to the input language of the model checking tool, model checking works for almost all kinds of properties. In contrast, within the context of this thesis, only safety properties like "a certain state s can never be reached" will be considered, where s represents some undesired or safety critical state. With regard to fault-tolerance and robustness, safety properties are most important. Other properties, like for example liveness properties, can be translated to safety properties using the approach of (Artho *et al.*, 2002).

Different model checking approaches have evolved within the last years, each of them having different advantages and disadvantages. Efficiency and quality of results of the different approaches thereby strongly depend on the system intended to be checked.

The most simple approach is called *explicite state model checking*: Every state of the system intended to be checked is represented explicitely. The complexity of model checking thus is linear in the size of the underlying system. This technique has been state of the art for quite a long time, but works well only for rather small systems. The number of applications using this technique therefor shrinks.

In contrast to that, the *symbolic model checking* (McMillan, 1993) uses data structures able to represent sets of states. That means, when working with this approach, the transition relation of the system is not explicitely constructed but represented by a Boolean function, for example. Other Boolean functions are used to represent sets of global states[1]; these are combined to so-called *regions*, one region containing states with similar clock valuations. The advantage in comparison with an explicite representation of the global states is that space requirements for Boolean functions are exponentially smaller. Thus, it is possible to model check larger systems.

Bounded Model Checking (Biere *et al.*, 1999b; Clarke *et al.*, 2001) works with a finite unravelling of the (explicite representation of the) transition relation. The next-state relation is encoded as a propositional formula, which is unrolled up to some finite given depth k. Unrolling in this context means to encode the whole transition relation (and thus all possible behaviour) for the first k steps.

In most model checking approaches, a combination of symbolic and bounded model checking is used. In this way, it it possible to benefit from both, as systems often are very large, but the property to be checked turns out to be fulfilled or not after a (compared to the system's size) quite small number of steps.

One of the main problems in model checking, however realised, is the so-called *state explosion problem*: Even though the system description of the system to be checked is finite, applying model checking may lead to infinitely many global states which have to be considered. This may be due variables with infinite range, as well as possible parallel execution of different system parts.

1.2. Abstraction Refinement

Abstraction refinement is one of the major approaches to cope with the state explosion problem, while trying to preserve efficiency and correctness of verification results.

In general, abstraction means to remove information from a system which is not relevant to the property to be verified. Due to this release of constraints, the set of reachable global states of the abstract system yields a superset of the set of reachable global states of the concrete system. In this way, if a specified safety property holds in the abstract system, it will hold in the concrete system, too.

One of the best know techniques for abstraction is *predicate abstraction* (Graf &

[1] A global state represents the complete status of the system at some point in time, that is not only the state the system is in, but also values of variables.

Saïdi, 1997): A set of initial predicates is chosen on the concrete states. Thereby, a predicate may be any Boolean expression over the variables of the concrete system, that means for example a constraint on state variables, program counters or clocks. The abstract state space induced by the set of predicates is given by the truth values of the predicates. Thus, for n predicates, the abstract state space has 2^n states.

As mentioned above, if the property holds in the abstract system, it will hold in the concrete system, too. On the other hand, if the property does not hold in the abstract system, nothing is known about the concrete system, as it is not sure whether the property does not hold at all or whether the abstraction has been "false". To detect this cause (why the property does not hold in the abstract system), refinement comes into play: Parameters which have been abstracted have to be taken into account again (they have to be refined). In case a counterexample has been found in the abstract system, but the property nevertheless is satisfied in the concrete system, the counterexample is said to be *spurious*. The main task in automatic abstraction refinement is to be able to identify parameters which are not relevant for the property (to abstract them) and which caused a counterexample to be spurious (to refine them), respectively.

1.3. Overview

The aim of this thesis is to verify safety properties of timed automata using SAT-based verification methods and bounded model checking. Safety properties are properties of the form "a certain state s is not reachable" (therefor, they will also be called reachability properties). The automaton and property are therefor translated into some formula representation providing the same behaviour as the automaton. To be able to cope with the state explosion problem, abstraction techniques are applied to the automata to reduce the number of states. Based on spurious counterexamples, the abstraction is refined and model checked again until either the property is shown to be valid within the actual bound, or a counterexample has been identified.

Figure 1.1 gives an overview of the conceptual design of this thesis. Grey components represent external, already existing tools, while all other "boxed" components represent different concepts which will be developed, explained and proven within the context of this thesis. Arrows represent data flow between the different components, with double head arrows represent data exchange between different tools. Keeping the aim of verifying reachability properties in mind, figure 1.1 will now be explained in detail, starting in the top left corner.

The model checker Uppaal (uppaal, 2005) is intended to be used as an editor. It provides a simple and intuitive modelling environment for creation of timed automata as well as reachability properties. The Representer is used to translate the timed automaton and the property obtained from Uppaal into the formula representation the following components will work on.

The part of the Abstracteur is to modify the formula representation of the timed automaton such that some parameters are removed from the system, that means these parameters are abstracted. In case one or more of these parameters intended to be ignored is contained in the property, they will be abstracted from the property,

too. Up to this point, the formula representation of the timed automaton describes its general behaviour, independent from a specific number of steps. But as bounded model checking is intended to be used (that means the behaviour of the automaton will be examined for the first k steps, with k being the bound), the automaton has to be *unrolled*. As described above, unrolling means to instantiate the general formula representation with concrete values for the first k steps. This is what the Unroller component does.

The formula representation of the unrolled automaton and the property will be given to some external SAT solver (decision procedure DP). Apart from the need to translate the formula representation into the input format for the SAT solver (and the output of the SAT solver into the formula representation back again), the DP component may be an arbitrary SAT solver. In case the SAT solver determines that state s is not reachable, the property is satisfied, at least for the first k steps. Further analysis may consist of increasing the unrolling depth. If instead s is reachable in the abstract automaton (that means the property is not satisfied at first glance), the DP component will return a trace through the automaton which finally reaches s. For this trace, it has to be checked whether it can be reconstructed in the concrete automaton or not. In the former case, the property does not hold (at least for bound k), while in the latter case, a spurious counterexample has been found.

The task of the Concretizer is to "translate back" the trace into the notation of the concrete automaton: Due to fewer constraints, one trace in the abstract automaton may correspond to a set of traces in the concrete automaton. The Concretizer identifies the set of concrete traces and adds appropriate constraints to the formula representation characterising this set. The concrete automaton, together with the concretized trace and the property then is unrolled again, and the external tool FOCI[2] (FOCI, 2005) is called.

FOCI is not only a SAT solver but an interpolating theorem prover, that means it derives craig interpolants from a proof of unsatisfiability. However, first of all, FOCI applies model checking to the concrete formula representation. In case the result is "s is reachable", the counterexample obtained from the DP component is a real counterexample, and the property is not satisfied. But in case FOCI returns "s is not reachable", the counterexample is spurious, and the question arises why. In more detail, the challenging question is why the counterexample has been possible in the abstract automaton, but not in the concrete one. Surely, one of the parameters which have been abstracted is "responsible".

Based on the craig interpolants obtained from FOCI, the Refiner tries to identify the cause of the spurious counterexample. Based on this cause, the Refiner undoes the abstraction of one or more parameters, in adding them to the formula representation again. This is called refinement. As the detection of the cause is based on one counterexample, while the Refiner tries to derive information about the parameters in general, refinement is always based on heuristics. The abstraction refinement loop then starts again with the reduced set of parameters to abstract from.

[2]Note that for convenience, FOCI is used within the DP component, too.

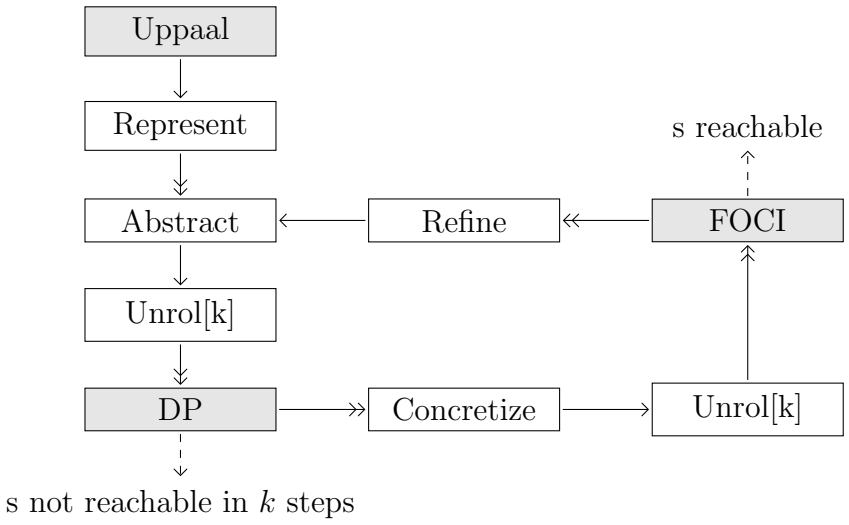


Figure 1.1.: System Architecture for SAT-based Abstraction Refinement Loop

1.3.1. Outline

The rest of this thesis is structured as follows: Chapter 2 will provide the theoretical background important for this thesis. In chapter 3, the formula representation of timed automata is introduced, together with the proof of correctness (with regard to the underlying timed automaton). Chapter 4 defines the abstraction technique used within the context of this thesis. As abstraction works on formulae, a main part of chapter 4 is the proof that abstraction preserves timed automata. That means: The abstraction of the formula representation of a timed automaton results in the formula representation of another timed automaton.

Chapter 5 and 6 present the basic concepts of craig interpolants, discuss their expressiveness and finally provide refinement heuristics.

All the aforementioned concepts have been implemented: Chapter 7 presents he resulting tool SAAtRe .

2. Fundamentals

2.1. Preliminaries

As the name implies, SAT solver tools—which are fundamental for this thesis—work with propositional formulae, and detect whether they are satisfiable or not. For this reason, a short introduction to the terminology of (propositional) logic as used within this theses will be given, essentially following (Schöning, 1995).

Let $\mathcal{P}$ be a countable set of symbols with typical elements $p, p_1, p_2, \ldots$, so-called (atomic) propositions, let $\mathcal{V}$ be a countable set of variables with typical elements x and y, let $\mathcal{S}_l = \{\wedge, \vee, \neg\}$ be the set of "basic" logical symbols and let $\mathcal{S}_a = \{+, -, *, /, <, \leq, =, \geq, >\}$ be the set of arithmetical symbols. $\mathcal{P}$, $\mathcal{V}$, $\mathcal{S}_l$ and $\mathcal{S}_a$ are pairwise dissimilar. Let `true` and `false` be two dissimilar symbols not already contained in $\mathcal{P}$, that means $\texttt{true}, \texttt{false} \notin \mathcal{P}, \texttt{true} \neq \texttt{false}$.

Definition 2.1.1 (Terms)
For $\mathcal{V}$ being a countable set of variables, the set $\mathcal{T}(\mathcal{V})$ of *terms over* $\mathcal{V}$ —with typical elements T, T_1, T_2—is defined inductively:

1. If $x \in \mathcal{V}$, then $x \in \mathcal{T}(\mathcal{V})$
2. If $T_1, T_2 \in \mathcal{T}(\mathcal{V})$, then $(T_1 + T_2) \in \mathcal{T}(\mathcal{V})$,
3. If $T_1, T_2 \in \mathcal{T}(\mathcal{V})$, then $(T_1 - T_2) \in \mathcal{T}(\mathcal{V})$,
4. If $T_1, T_2 \in \mathcal{T}(\mathcal{V})$, then $(T_1 * T_2) \in \mathcal{T}(\mathcal{V})$,
5. If $T_1, T_2 \in \mathcal{T}(\mathcal{V})$, then $(T_1/T_2) \in \mathcal{T}(\mathcal{V})$,
6. $\mathcal{T}(\mathcal{V})$ is the minimal set with those properties

For $T \in \mathcal{T}(\mathcal{V})$, let $Var(T) \subset \mathcal{V}$ be the *set of variables in* T, that means the (finite) subset of $\mathcal{V}$ containing all variables appearing in T. □

Definition 2.1.2 (Formulae)
For $\mathcal{V}$ being a countable set of variables and $\mathcal{P}$ being a countable set of atomic propositions, the set $\mathcal{F}(\mathcal{P}, \mathcal{V})$ of *formulae over* $\mathcal{P} \cup \mathcal{V}$—with typical elements F, F_1, F_2—is defined inductively:

1. If $T_1, T_2 \in \mathcal{T}(\mathcal{V})$, then $(T_1 \sim T_2) \in \mathcal{F}(\mathcal{P}, \mathcal{V})$, for $\sim \in \{<, \leq, =, \geq, >\}$
2. If $p \in \mathcal{P}$, then $p \in \mathcal{F}(\mathcal{P}, \mathcal{V})$
3. $\texttt{true} \in \mathcal{F}(\mathcal{P}, \mathcal{V})$

4. `false` $\in \mathcal{F}(\mathcal{P}, \mathcal{V})$

5. If $F \in \mathcal{F}(\mathcal{P}, \mathcal{V})$, then $\neg F \in \mathcal{F}(\mathcal{P}, \mathcal{V})$

6. If $F_1, F_2 \in \mathcal{F}(\mathcal{P}, \mathcal{V})$, then $(F_1 \wedge F_2) \in \mathcal{F}(\mathcal{P}, \mathcal{V})$

7. If $F_1, F_2 \in \mathcal{F}(\mathcal{P}, \mathcal{V})$, then $(F_1 \vee F_2) \in \mathcal{F}(\mathcal{P}, \mathcal{V})$

8. $\mathcal{F}(\mathcal{P}, \mathcal{V})$ is the minimal set with those properties

For $F \in \mathcal{F}(\mathcal{P}, \mathcal{V})$, let $Var(F) \subset \mathcal{V} \cup \mathcal{P}$ be the *set of variables in* F, that means the (finite) subset of $\mathcal{V} \cup \mathcal{P}$ containing all variables appearing in F. □

The symbols $\rightarrow$ (*implication*) and $\leftrightarrow$ (*equivalence*) are seen as abbreviations in the usual way, that means

$$F_1 \rightarrow F_2 \stackrel{def}{=} \neg F_1 \vee F_2$$
$$F_1 \leftrightarrow F_2 \stackrel{def}{=} (F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2).$$

To avoid ambiguities and for readability (to avoid parenthesis), $\neg$ has the highest priority, and $\wedge$ and $\vee$ have a higher priority than $\rightarrow$ and $\leftrightarrow$. The latter two are of the same priority, and so are $\wedge$ and $\vee$. The priorities of the arithmetic operators are as usual, that means

- $*$ and $/$ are of the same priority, which is higher than the one of $+$ and $-$
- $+$ and $-$ are of the same priority, which is higher than the one of $<, \leq, =, \geq$ and $>$
- $<, \leq, =, \geq$ and $>$ all have the same priority

Furthermore, all binary operators in $\mathcal{S}_l$ and $\mathcal{S}_a$ as well as $\rightarrow$ and $\leftrightarrow$ are left associative.

Some more basic notions will now be introduced for use in the sequel.

Formulae built without item 1 in definition 2.1.2 are called *propositional formulae*. An *atomic formula* is a formula built only with items 1, 2, 3 and 4 in definition 2.1.2. A *literal* is an atomic formula or its negation (thus built with items 1, 2, 3, 4 and 5 in definition 2.1.2). With respect to an atomic formula a, the literal a is called *positive literal*, while $\neg a$ is called *negative literal*. A *clause* is a finite disjunction of pairwise dissimilar literals, that means a formula such as

$$p_1 \vee p_2 \vee \ldots \vee p_n \vee \neg p'_1 \vee \neg p'_2 \vee \ldots \vee \neg p'_m,$$

with p_i, p'_j being atomic formulae. For $n = m = 0$, the clause it called *empty clause*, written as $\square$, and is semantically equivalent to `false`.

Normal forms can be defined for formulae, expressing the fact that the formula has a special structure.

Definition 2.1.3 (Normal Forms)
Let F be a formula, let $\{l_1, \ldots, l_k\}$ be a set of literals. F is considered to be in *conjunctive normal form (CNF)* if it is a conjunction of clauses, that means if it is of the form

$$F = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} l_{i,j},$$

with $l_{i,j} \in \{l_1, \ldots, l_k\}$ for all i, j. Let $\mathcal{F}(\mathcal{P}, \mathcal{V})_{cnf}$ be the set of formulae which are in CNF.

F is considered to be in *disjunctive normal form (DNF)* if it is a disjunction of conjunctions of literals, that means if it is of the form

$$F = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} l_{i,j},$$

with $l_{i,j} \in \{l_1, \ldots, l_k\}$ for all i, j. Let $\mathcal{F}(\mathcal{P}, \mathcal{V})_{dnf}$ be the set of formulae in DNF.

F is considered to be in *negation normal form (NNF)* if it only contains the logical operators $\wedge, \vee$ and $\neg$, and negations appear only in front of literals. Let $\mathcal{F}(\mathcal{P}, \mathcal{V})_{nnf}$ be the set of formulae in NNF.

All logical formulae can be converted into an equivalent formula in CNF (see appendix A.1.3), DNF (see appendix A.1.4) and NNF (see appendix A.1.2) through elimination of double negation and repeated application of the distributive law and de Morgan's rules (see appendix A.1.1), although this may lead to an exponential explosion of the formula. □

Most SAT solver work with formulae in CNF. The advantage in using a CNF formula representation is that—for the formula to be satisfied—all clauses have to be satisfied (that means have to evaluate to `tt`). This is fulfilled if within every clause, at least one literal is interpreted as `tt`. If one such literal is found for a clause, the other literals are no longer important for the satisfiability of the formula. Proofs using formula in CNF are therefor more efficient, and most algorithms working on CNF are much more efficient than algorithms working on arbitrary formulae. Furthermore, CNF is needed for resolution proofs (see definition 2.2.1 on page 15).

Remark 2.1.4 (CNF as Set of Clauses)
As a formula in CNF is a conjunction of clauses by definition, it is possible to unambiguously identify the formula by the set of clauses it is built from. Equivalently, an arbitrary formula (which is not necessarily in CNF) may be unambiguously identified by the set of its conjunctive elements. Therefor, for convenience, the notations shall be seen as equal. □

Example 2.1.5 (CNF, DNF and NNF)
Consider the formulae

$$F_1 = (\neg p_1 \vee p_2) \wedge (\neg p_2 \vee p_1)$$

and

$$F_2 = (p_1 \wedge p_2) \vee (\neg p_2 \wedge \neg p_1)$$

The formulae are semantically equivalent, formula F_1 is in CNF, while F_2 is in DNF. Both formulae are in NNF.

It is also possible to denote F_1 as $\{(\neg p_1 \vee p_2), (\neg p_2 \vee p_1)\}$ (see remark 2.1.4). □

All simple conjunctions[1] are in CNF, as they are a conjunction of clauses containing only one literal each. But also all simple disjunctions are in CNF, as they are a conjunction of a single clause. With the same argument, all simple conjunctions and disjunctions are in DNF.

Mappings are used in various cases, thus, some notations for mappings are introduced.

Remark 2.1.6 (Notations for Mappings)
For two mappings $f : A \to B$ and $f' : B' \to C$, with $B \subseteq B'$, the *composition* $f' \circ f$ is defined as the consecutive application of f and f', denoted as $f'(f(a))$:

$$\begin{aligned}(f' \circ f) &: A \to C\\ a &\mapsto f'(f(a))\end{aligned}$$

For two mappings $\sigma_1 : A \to B$ and $\sigma_2 : A' \to B'$, with $A \cap A' = \emptyset$, the *coproduct of* σ_1 *and* σ_2, denoted by $\sigma_1 \oplus \sigma_2$, is defined as

$$(\sigma_1 \oplus \sigma_2)(v) = \begin{cases} \sigma_1(v), v \in A \\ \sigma_2(v), v \in A' \end{cases}$$

A special mapping is the *identity*, denoted by *id*, which maps every element to itself:

$$\begin{aligned}id &: A \to A\\ a &\mapsto a\end{aligned}$$

□

Substitutions are special mappings, which are going to be used when defining the formula representation of timed automata (definition 3.3.11).

Definition 2.1.7 (Substitution)
Let $\mathcal{V}$ be a set of variables, and let $\mathcal{P}$ be a set of symbols. A *substitution* Θ is a finite mapping

$$\begin{aligned}&\Theta : \mathcal{V} \cup \mathcal{P} \to \mathcal{T}(\mathcal{V}) \cup \mathcal{F}(\mathcal{V}, \mathcal{P})\\ &\Theta = \{v_1/T_1, \ldots, v_n/T_n, p_1/F_1, \ldots, p_m/F_m\}, (n, m \in \mathbb{N})\end{aligned}$$

from variables and propositional symbols to terms and propositional formulae, respectively, with

- $v_1, \ldots, v_n \in \mathcal{V}$ are pairwise dissimilar variables
- $T_1, \ldots, T_n \in \mathcal{T}(\mathcal{V})$ are terms
- $p_1, \ldots, p_m \in \mathcal{P}$ are pairwise dissimilar symbols
- $F_1, \ldots, F_m \in \mathcal{F}(\mathcal{V}, \mathcal{P})$ are formulae

[1] A simple conjunction is a formula like $p_1 \wedge p_2 \wedge \neg p_3$, that means a conjunction of literals.

For X being a term or formula and

$$\Theta = \{v_1/T_1, \ldots, v_n/T_n, p_1/F_1, \ldots, p_m/F_m\}$$

being a substitution, the *application of* Θ *to* X, denoted by $X\Theta$, is defined inductively on the structure of X. Let v and p be a variable and a symbol, respectively, let $T_1, T_2 \in \mathcal{T}(\mathcal{V})$ be terms and let $F_1, F_2 \in \mathcal{F}(\mathcal{P}, \mathcal{V})$ be formulae.

$$\begin{aligned}
v\Theta &= \begin{cases} T_i & \text{for } v = v_i \text{ for some } i \in \{1, \ldots, n\} \text{ and } (v_i/T_i) \in \Theta \\ v, & \text{otherwise} \end{cases} \\
p\Theta &= \begin{cases} F_i, & \text{for } p = p_i \text{ for some } i \in \{1, \ldots, m\} \text{ and } (p_i/F_i) \in \Theta \\ p, & \text{otherwise} \end{cases} \\
(T_1 \sim T_2)\Theta &= T_1\Theta \sim T_2\Theta, \quad \text{for } \sim \in \{+, -, *, /\} \cup \{<, \leq, =, \geq, >\} \\
(F_1 \sim F_2)\Theta &= F_1\Theta \sim F_2\Theta, \quad \text{for } \sim \in \{\wedge, \vee\} \\
(\neg F_1)\Theta &= \neg(F_1\Theta)
\end{aligned}$$

A single element $(x_i/X_i) \in \Theta$ is called *binding* and means $\Theta(x_i) = X_i$. For x_i being a variable and X_i being a term, X_i is said to be *bound* to x_i, and x_i is said to be *instantiated* by X_i. For x_i being a symbol and X_i being a formula, the parlance is defined in the same way.

For two substitutions

$$\begin{aligned}
\Theta_1 &= \{x_1/X_1, \ldots, x_n/X_n\} \text{ and} \\
\Theta_2 &= \{y_1/Y_1, \ldots, y_m/Y_m\},
\end{aligned}$$

the *composition of* Θ_1 *and* Θ_2, denoted by $\Theta_1\Theta_2$, is obtained from the set

$$\{x_1/X_1\Theta_2, \ldots, x_n/X_n\Theta_2, y_1/Y_1, \ldots, y_m/Y_m\}$$

by removing all bindings

- y_j/Y_j with $y_j \in \{x_1, \ldots, x_n\}, j \in \{1, \ldots, m\}$
- $x_i/X_i\Theta_2$ with $X_i\Theta_2 = x_i, i \in \{1, \ldots, n\}$

□

Note that the application of a substitution

$$\Theta = \{v_1/T_1, \ldots, v_n/T_n, p_1/F_1, \ldots, p_m/F_m\}$$

to a term or variable is well-defined due to the fact that $v_1, \ldots, v_n$ and $p_1, \ldots, p_m$ are pairwise dissimilar.

For $\mathcal{V}$ being a countable set of variables and $x_i \in \mathcal{V}$, let D_i be the range (set of possible values) of x_i. D_i may be finite as well as infinite. Let D be the union of all ranges of variables in $\mathcal{V}$,

$$D = D_1 \cup D_2 \cup D_3 \cup \ldots.$$

Definition 2.1.8 (Interpretation)
An *interpretation* is a mapping from variables to a range or a set of ranges:

$$\sigma : \mathcal{V} \to D,$$

assigning to every variable $v \in \mathcal{V}$ a value out of its range $D_i \subseteq D$. The class of possible interpretations for a given set of variables $\mathcal{V}$ is denoted by $Int(\mathcal{V})$. □

The *semantics* of terms and formulae is conceptually defined via two interpretations I and β. For a proposition $p \in \mathcal{P}$, I is a mapping from $\mathcal{P}$ to the set $\mathbb{B}$of Boolean values, that is

$$I : \mathcal{P} \to \mathbb{B},$$

with $\mathbb{B} = \{\mathtt{tt}, \mathtt{ff}\}$. $\mathtt{tt}$ denotes the Boolean value "1", sometimes written as "true" or "$\top$", while $\mathtt{ff}$ denotes the Boolean value "0", sometimes written as "false" or "$\bot$". I is also called a *truth-value interpretation*. For a variable $x_i \in \mathcal{V}$, β assigns a value $d_i \in D_i$ to it, that is

$$\beta : x_i \to D_i \text{ for every } x_i \in \mathcal{V}.$$

β is called a *variable assignment*.

$I(p)$ denotes the truth value of proposition p under the interpretation I, and $\beta(x)$ denotes the value of variable x under the interpretation β, respectively. For a proposition p, we also write $I \models p$ for $I(p) = \mathtt{tt}$ and $I \not\models p$ for $I(p) = \mathtt{ff}$, respectively.

Definition 2.1.9 (Semantics of Terms)
The semantics of terms

$$\mathcal{I}[\![T]\!](\sigma) : \mathcal{T}(\mathcal{V}) \to D$$

with regard to an interpretation $\sigma = I \oplus \beta, \sigma \in Int(\mathcal{V} \cup \mathcal{P})$ being the coproduct of a truth-value interpretation I and a variable assignment β is defined inductively as

1. $\mathcal{I}[\![x]\!](\sigma) = \beta(x)$
2. $\mathcal{I}[\![T_1 + T_2]\!](\sigma) = \mathcal{I}[\![T_1]\!](\sigma) + \mathcal{I}[\![T_2]\!](\sigma)$
3. $\mathcal{I}[\![T_1 - T_2]\!](\sigma) = \mathcal{I}[\![T_1]\!](\sigma) - \mathcal{I}[\![T_2]\!](\sigma)$
4. $\mathcal{I}[\![T_1 * T_2]\!](\sigma) = \mathcal{I}[\![T_1]\!](\sigma) * \mathcal{I}[\![T_2]\!](\sigma)$
5. $\mathcal{I}[\![T_1 / T_2]\!](\sigma) = \mathcal{I}[\![T_1]\!](\sigma) / \mathcal{I}[\![T_2]\!](\sigma)$

□

Definition 2.1.10 (Semantics of Formulae)
The semantics of formulae

$$\mathcal{I}[\![F]\!](\sigma) : \mathcal{F}(\mathcal{P}, \mathcal{V}) \to \mathbb{B}$$

with regard to an interpretation $\sigma = I \oplus \beta, \sigma \in Int(\mathcal{V} \cup \mathcal{P})$ being the coproduct of a truth-value interpretation I and a variable assignment β is defined inductively as

1. $\mathcal{I}[\![T_1 \sim T_2]\!](\sigma) = \texttt{tt}$ iff $(\mathcal{I}[\![T_1]\!](\sigma) \sim \mathcal{I}[\![T_2]\!](\sigma)) = \texttt{tt}$, for $\sim \in \{<, \leq, =, \geq, >\}$
2. $\mathcal{I}[\![p]\!](\sigma) = I(p)$
3. $\mathcal{I}[\![\texttt{true}]\!](\sigma) = \texttt{tt}$
4. $\mathcal{I}[\![\texttt{false}]\!](\sigma) = \texttt{ff}$
5. $\mathcal{I}[\![\neg F]\!](\sigma) = \texttt{tt}$ iff $\mathcal{I}[\![F]\!](\sigma) = \texttt{ff}$
6. $\mathcal{I}[\![F_1 \wedge F_2]\!](\sigma) = \texttt{tt}$ iff $(\mathcal{I}[\![F_1]\!](\sigma) = \texttt{tt}$ and $\mathcal{I}[\![F_2]\!](\sigma) = \texttt{tt})$
7. $\mathcal{I}[\![F_1 \vee F_2]\!](\sigma) = \texttt{tt}$ iff $(\mathcal{I}[\![F_1]\!](\sigma) = \texttt{tt}$ or $\mathcal{I}[\![F_2]\!](\sigma) = \texttt{tt})$

□

Note that `true` and `false` are constants introduced for convenience, denoting the formulae which always evaluate to `tt` and `ff`, respectively.

An interpretation σ *satisfies* a formula F—written as $\sigma \models F$—iff $\mathcal{I}[\![F]\!](\sigma) = \texttt{tt}$. σ is called *model of* F iff $\sigma \models F$. F is called *satisfiable* iff there exists an interpretation σ such that $\mathcal{I}[\![F]\!](\sigma) = \texttt{tt}$, otherwise it is called *unsatisfiable*. F is called *valid*—written as $\models F$—iff all interpretations σ (of its signature) satisfy it.

Definition 2.1.11 (Corresponding Signature)
Let $S = \{T_1, \ldots, T_n, F_1, \ldots, F_m\}$, $n, m \geq 0$, be a set of terms T_i and formulae F_j. The *corresponding signature* $\Sigma(S)$ *of* S is the set of variables appearing in the terms and formulae of S:

$$\Sigma(S) \stackrel{def}{=} Var(T_1) \cup \ldots \cup Var(T_n) \cup Var(F_1) \cup \ldots \cup Var(F_m)$$

□

Definition 2.1.12 (Calculus)
Let M be a set of formulae. A *calculus* or *proof system* $\mathcal{C}$ on M is a finite set of *(proof) rules*, each rule being a decidable relation on M. The set notation for a single proof rule R is

$$R = \{(F_1, \ldots, F_n, F_{n+1}) \mid \text{ where } B(F_1, \ldots, F_{n+1})\},$$

with F_i $(1 \leq i \leq n+1)$ being schemes for formulae which have to fulfil the decidable condition B. $F_1, \ldots, F_n$ are called the *premises* of R, F_{n+1} is called the *conclusion* of R.

In addition, the following syntax will be used:

$$R : \frac{F_1, \ldots, F_n}{F_{n+1}}, \text{ where } B(F_1, \ldots, F_{n+1}),$$

A proof rule without premises is called *axiom*. □

Definition 2.1.13 (Calculus Proof)
Let F be a formula, $\mathcal{G}$ a set of formulae and $\mathcal{C}$ a calculus. A *proof of* F *from premises* $\mathcal{G}$ *in* $\mathcal{C}$, denoted as $\mathcal{G} \vdash_{\mathcal{C}} F$ (or just $\mathcal{G} \vdash F$), is a finite sequence of formulae $G_1, \ldots, G_n, F$, such that every formula G_i $(1 \leq i \leq n)$ is

1. either a premise (that means contained in $\mathcal{G}$: $G_i \in \mathcal{G}$) or
2. an axiom of $\mathcal{C}$ (that means a rule without premises) or
3. there exists a proof rule $R \in \mathcal{C}$ and formulae $G_{i_1}, \ldots, G_{i_k}$ $(i_1, \ldots, i_k < i)$ such that $R = \{(G_{i_1}, \ldots, G_{i_k}, G_i)\}$.

□

A formula G_i obtained from item 3 in definition 2.1.13 is said to be the *result* or *conclusion of applying proof rule R to premises $G_{i_1}, \ldots, G_{i_k}$*, F is called the *conclusion of the proof*.

For readability, calculus proofs may be represented as a tree. As a single proof rule $R = \{(G_1, \ldots, G_k, G_{k+1})\}$ of calculus $\mathcal{C}$ represents a proof of G_{k+1} from $G_1, \ldots, G_k$ in $\mathcal{C}$, the tree representation is also defined for proof rules.

Remark 2.1.14 (Tree Representation of a Calculus Proof)
Let F be a formula, $\mathcal{G}$ be a set of formulae and $\mathcal{C}$ be a calculus. Let $(G_1, \ldots, G_n, F)$ be a proof of F from premises $\mathcal{G}$ in $\mathcal{C}$. The *tree representation* of this proof is a finite ordered tree, constructed from the proof as follows:

- For every G_i in $(G_1, \ldots, G_n, F)$ which is a premise obtained from $\mathcal{G}$ (item 1 in definition 2.1.13), there exists a unique leaf node labelled with G_i
- For every G_i in $(G_1, \ldots, G_n, F)$ which is an axiom obtained from $\mathcal{C}$ (item 2 in definition 2.1.13), there exists a unique leaf node labelled with G_i
- For a set of formulae $\{G_{i_1}, \ldots, G_{i_k}\}$ in $(G_1, \ldots, G_n, F)$ and a proof rule $R = \{(G_{i_1}, \ldots, G_{i_k}, G_i)\}$, $(i_1, \ldots, i_k < i)$ with premises $G_{i_1}, \ldots, G_{i_k}$ and a formula G_i such that G_i is obtained from applying R to $G_{i_1}, \ldots, G_{i_k}$, there exists a unique node labelled with G_i which is successor of all nodes labelled with $G_{i_1}, \ldots, G_{i_k}$. In case $G_i = F$, this node is the root node of the tree, otherwise it is an inner node.

□

For an example of the tree representation of a calculus proof, please consider example 2.2.2 on the next page.

2.2. Resolution

In 1965, J.A. Robinson presented a new method to combine the up to that day not directly associated approaches in automatic theorem-proving[2] - substitution and truth-functional analysis of the results - into a new one: The *resolution calculus* (Robinson, 1965). The calculus is based on just one inference rule, the *resolution rule*, which particularly means it does not contain any axioms. The basic resolution principle works with formulae in CNF, represented as clause sets:

[2]Automatic theorem-proving means usage of computers to deduce logical formulae from premises.

Definition 2.2.1 (Propositional Resolution Rule)
Let $A = (A_1 \vee A_2 \vee \dots A_n)$ and $B = (B_1 \vee B_2 \vee \dots B_m)$ be two clauses, and L be a literal. The *propositional resolution rule* is

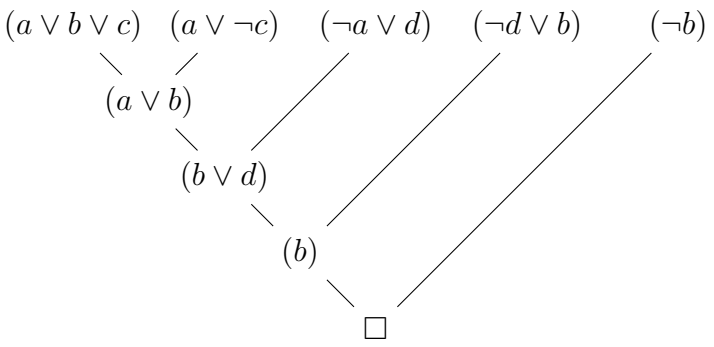
$$\frac{A \vee L,\ B \vee \neg L}{A \vee B}$$

The conclusion of the resolution rule is called *resolvent*. A proof of $A \vee B$ from $A \vee L$ and $B \vee \neg L$ is also called *resolution (step) on* L. □

Robinson proved the calculus to be sound (that means resolvents are logical consequences of the premises[3]) and refutation complete. Refutation complete means: Whenever two formulae A and B are unsatisfiable, a contradiction (the empty clause in this case) is derivable by resolution. In this way, the calculus is simple but powerful.

In general, resolution is used to check formulae for satisfiability and validity. At its actual meaning, resolution shall only be used to show unsatisfiability, as satisfiability for arbitrary formulae is not decidable[4]. Nevertheless, it is possible to show validity by proving the unsatisfiability of the negation.

Example 2.2.2 (Resolution Proof, Tree Representation)
Consider the set of propositional clauses $\{(a \vee b \vee c), (a \vee \neg c), (\neg a \vee d), (\neg d \vee b), (\neg b)\}$. The set can be proved to be inconsistent by the following resolution proof:



□

2.3. SAT Solver Tools

2.3.1. Overview

Given a propositional formula, the problem whether there exists a satisfying assignment such that the formula evaluates to `tt` is called "Boolean Satisfiability Problem (SAT)". A SAT solver thus is a tool deciding whether the formula is satisfiable or not, and, in case it is, returns a satisfying assignment. Since SAT is an NP-complete

[3]It is quite obvious that every valuation satisfying the premises will also satisfies the resolvent.
[4]For a propositional formula, satisfiability is decidable with at most exponential complexity.

problem (Cook, 1971) with complexity in $O(n^2)$ in the worst case (n being the number of different variables appearing in the formula)[5], a main task when designing a SAT solver is to improve efficiency (see for example (Moskewicz *et al.*, 2001)).

For this reason, most SAT solver are based on the Davis-Putnam search algorithm (Davis & Putnam, 1960) or the Davis-Loveland-Logemann search algorithm (Davis *et al.*, 1962), the latter being an improved version of the former. Both of them work with resolution, and thus with formulae in CNF. The formula to be checked is unsatisfiable if the empty clause can be derived from the original clauses. As both algorithms have great space requirements, they are combined with heuristic local search (Aarts & Lenstra, 1997) or methods for boolean constraint propagation (Apt, 2000) in most implementations.

In *heuristic local search* techniques, a neighbouring function is defined on the set of possible valuations for the variables contained in the formula, denoting the fact that a certain valuation is "close" to another one (for example by switching the value of only one variable). Furthermore, the neighbouring function denotes which valuation of the two is "better" with regard to the satisfiability of the formula. Both relations of being close and being better are heuristics and strongly depend on the formula.

Heuristic local search works iteratively: Starting from some initial valuation, the algorithm checks whether there exists a valuation which is close the actual one but better. If so, this one is taken over and the search for a better valuation starts again. Further heuristics are needed to avoid being stuck with some unsatisfying valuation although the formula itself is satisfiable (local optima).

Boolean constraint propagation is a technique to propagate known constrains and restrictions on the variables of the formula as early as possible. Therefor, required valuations are deduced as early as possible: If all literals but one contained in a clause are set to `false` by the decision procedure, the remaining literal has to be set to `tt` for the formula to be satisfiable (only possible for formulae in CNF).

SAT solver nowadays become more and more important, as many practical problems may be reduced to Boolean formulae such that SAT solver are applicable. Application ranges are for example planning problems in artificial intelligence, automatic test generation, and as one of the major fields surely model checking and verification. Well-known publicly available SAT solver are for example GRASP (Marques-Silva & Sakallah, 1999), SATO (Zhang, 1997), Chaff (Moskewicz *et al.*, 2001) and FOCI.

2.3.2. The Interpolating Theorem Prover FOCI

FOCI is a command line based tool, developed by Kenneth L. McMillan (FOCI, 2005). The abilities of FOCI extend those of a pure SAT solver, in that it provides an implementation of first order craig interpolation for the combined theories of Boolean and numeric variables: FOCI is able to solve propositional formulae (containing disjunction, conjunction and negation) as well as linear (in)equalities on numeric

[5]The brute-force solution to solve the satisfiability problem is to simply try all possible assignments. These are n^2 possible assignments for n variables.

variables (containing the relational operators $=$, $\leq$ and the arithmetical operators $+$ and $*$). The input language of FOCI thereby uses a prefix notation.

Numeric variables are handled using either an integer or a rational interpretation, which one to use is declared by a command line option. This means in particular that it is not possible to use both numeric interpretations at the same time for variables in the same input file. Another command line option is whether to allow universally quantified variables in the interpolants or not. This is prohibited by default, and will not be used within this thesis. For more details on quantified interpolants, please refer to (McMillan, 2004), for example.

Using the integer interpretation, FOCI is complete only for separation formulae, that means formulae where the arithmetic expressions are of the form $x \leq y + c$ or $x \leq c$, with c being a numeric constant. For rational interpretation, FOCI is complete for arbitrary formulae. Within the context of this thesis, the rational interpretation will be used, see also section 3.2.

With regard to the interpolants obtained for a formula sequence, FOCI is able to perform symmetric and asymmetric interpolation. The asymmetric interpolation—which is going to be used here—will be presented in chapter 5. For further information about symmetric interpolation, please consider (McMillan, 2005a).

2.4. Timed Automata

Timed automata were introduced by R. Alur and D. L. Dill in 1994 (Alur & Dill, 1994) as an extension of Büchi-automata (which are again an extension of finite automata, see for example (Thomas, 1990)), to model the behaviour of real-time systems over time. When talking about time, there are mainly two choices for the time domain: Discrete time (with time values $\in \mathbb{N}$) and continuous time (with time values $\in \mathbb{R}$). While discrete time is close to hardware implementation of realtime systems (in that a discrete time step can be taken with every trailing edge, for example), continuous time is a more realistic approach (as real time indeed is continous). For this reason, only continuous linear time shall be considered here, that is

$$\texttt{Time} = \mathbb{R}_{\geq 0}.$$

As in Büchi-automata, the behaviour is represented by an infinite sequence of actions, which for timed automata is paired with an infinite sequence of (real-valued) times. Conceptually, this leads to a sequence of pairs, each pair consisting of an action and a time stamp, with the intended meaning that the action takes place at that time. In the automaton model presented here, there are two types of actions: visible and invisible actions. The visible actions are used for synchronisation of automata, while the invisible actions are used for an internal step of a single automaton (that means independent from other automata). The underlying idea is that transitions are instantaneous and time may only elapse when the automaton remains in one of its states.

For time measurement, every timed automaton has a finite set of clock variables (or simply clocks) ranging over `Time`. Transitions may only be taken if the current values of the clocks satisfy the associated constraints, and transitions may reset clocks to 0.

2.4.1. Syntax

Essentially following (Alur & Dill, 1994) and (Alur, 1999), the syntax of timed automata will now be defined formally.

Transitions of timed automata are associated with clock constraints (also called *guards*) restricting the possibilities to fire the transition, that means a transition may only be taken if the current values of the clocks satisfy the guard. An invariant is assigned to every state constraining the values of clocks for which the automaton may remain in this state.

Definition 2.4.1 (Guards, Invariants, Clock Constraints)
Let $X \subset \mathcal{V}$ be a finite set of real-valued variables, called *clocks*.

a) The set $\Phi_G(X) \subset \mathcal{F}(\emptyset, X)$ of *guards over* X is defined by the set of formulae which is built with items 1 and 3' in definition 2.1.1 on page 7 and items 1', 3, 4, 5, 6 and 7 in definition 2.1.2 on page 7, where item 1' is a slightly modified version of item 1 from 2.1.2 and 3' is a slightly modified version of item 3 from 2.1.1:

   1'. If $T_1 \in \mathcal{T}(X)$, $d \in \mathbb{Q}$, then $(T_1 \sim d) \in \mathcal{F}(\emptyset, X)$, for $\sim \in \{<, \leq, =, \geq, >\}$

   3'. If $x_1, x_2 \in \mathcal{V}$, then $(x_1 - x_2) \in \mathcal{T}(\mathcal{V})$

b) The set $\Phi_I(X) \subset \mathcal{F}(\emptyset, X))$ of *invariants over* X is defined by the set of formulae which is built with items 1 and 3' in definition 2.1.1 on page 7 and items 1', 3, 4 and 6 in definition 2.1.2 on page 7, with 1' and 3' as defined as above.

Both guards and invariants are called *clock constraints (over X)*, where $\Phi(X)$ denotes the set of all clock constraints. □

Thus, a guard φ may be described with the BNF-grammar

$$\varphi = x \sim c \mid x - y \sim c \mid \texttt{true} \mid \texttt{false} \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi_1,$$

while an invariant φ may be described with the BNF-grammar

$$\varphi = x \sim c \mid x - y \sim c \mid \texttt{true} \mid \texttt{false} \mid \varphi_1 \wedge \varphi_2,$$

for x, y being clocks, c being a rational constant and $\sim \in \{<, \leq, =, \geq, >\}$.

The restriction to rational time constants is needed for decidability of emptiness and reachability of timed automata (see for example (Alur & Dill, 1994) or (Alur, 1999)).

The intended meaning of invariants is: The automaton may remain in a state while time elapses, and as long as the clocks satisfy the invariant, but if the invariant is no longer fulfilled, the state has to be left immediately. State invariants may thus be used to force a transition, for example. To ensure this behaviour, invariants have to be *convex* (see also (Alur & Dill, 1994)), which is property going to be used in the formula representation, chapter 3. Convexity of invariants is a semantical property, so it will be shown later; please refer to lemma 2.4.7 on page 21.

Timed automata working in parallel may synchronise with each other via transitions.

Definition 2.4.2 (Action Set)
Let $\underline{A}$ be an alphabet. The *action set* $\underline{A}_{?!}$ belonging to $\underline{A}$ is defined as

$$\underline{A}_{?!} = \{a? \mid a \in \underline{A}\} \cup \{a! \mid a \in \underline{A}\} \cup \tau,$$

with $\tau \notin \underline{A}$. Elements of $\underline{A}_{?!}$ are called *visible actions* or just *actions* (with $a?$ being an *input action* and $a!$ being an *output action*), τ is called *internal (or invisible) action*. For two actions $a!$ and $a?$, a is called the *underlying channel*. □

Using this, timed automata may be defined as following.

Definition 2.4.3 (Timed Automaton)
A *timed automaton* $\mathcal{A}$ is a tuple $\mathcal{A} = (\underline{A}, S, s_0, X, I, E)$ with

- $\underline{A}$ is a finite alphabet, the set of *events* or sometimes called *channels*,
- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- X is a finite set of clocks,
- $I : S \to \Phi_I(X)$ is a mapping assigning an invariant to every state, and
- $E \subseteq S \times \underline{A}_{?!} \times \Phi_G(X) \times \mathcal{P}(\mathcal{X}) \times \mathcal{S}$ is the set of transitions. An element $(s, \alpha, \varphi, Y, s')$ describes an edge from state s to state s' on event α. The transition is labelled with the guard φ specifying the enabling constraint, and resets all the clocks in the set Y.

Let $\mathcal{TA}$ be the set of all timed automata. □

According to this definition, transitions are labelled either with a visible action (input or output) or with the internal (and hence invisible from outside) action. The intended meaning of visible actions is: Two transitions of different timed automata, where one is labelled with an input action and the other one with an output action of the same underlying channel may only fire simultaneously. The actions are thus used for synchronisation.

In contrast to that, the internal action τ is used to represent a transition independent from other automata. Thus, when working with a single automaton, all its transitions have to be labelled with the internal action. For readability, τ may be omitted (transitions are written as (s, φ, Y, s') then) as well as invariants and guards equal to `true`.

Example 2.4.4 (Timed Automaton)
Consider the timed automata in figure 2.1 on the next page, modelling an "intelligent" light controller, and a human pressing it. The automaton representing the human pressing the light switch consists of only one state and a loop labelled with press!, meaning the human presses the switch nondeterministically. The controller consist of three states, representing the three possible states of the light, and a clock x. In its initial state, the light is off. On event press (that means the "controller

automaton" executes a transition labelled with action press?, while the "human automaton" executes a transition labelled with action press!), the light switches on, and x—or, in more detail, all elements in the set $\{x\}$—is set to 0. If the light switch is pressed again within 3 seconds, the light becomes brighter (guaranteed by the guard $x \leq 3$). Otherwise, it is turned off.

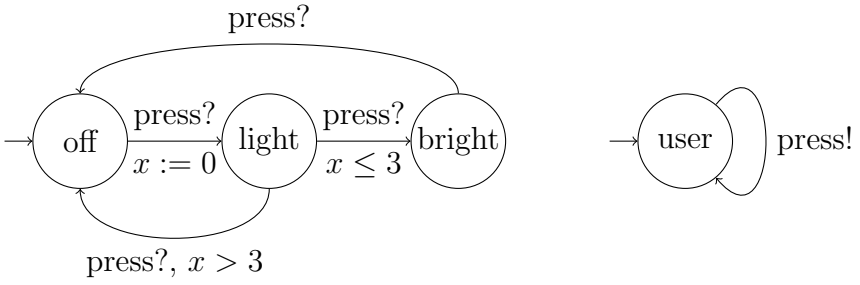


Figure 2.1.: Intelligent light controller

□

2.4.2. Semantics

To define the semantics of timed automata, first the concept of valuations for clocks is needed.

Definition 2.4.5 (Valuation)
Let X and Y be sets of clocks, with $Y \subseteq X$, let $x \in X$ be a clock and $t \in \texttt{Time}$. A *valuation ν of the clocks in X* is a mapping

$$\nu : X \to \texttt{Time}$$

assigning a real value to each clock. $\nu(x)$ denotes the actual value of x under the valuation ν. Val_X denotes the set of all possible valuations for the set of clocks X,

$$Val_X = X \to \texttt{Time}.$$

The *restriction of ν from X to Y*, denoted by $\nu|_Y$, for a clock x is defined by

$$\nu|_Y(x) = \begin{cases} \nu(x), & \text{if } x \in Y \\ \text{undefined}, & \text{if } x \in X \setminus Y \end{cases}$$

That means for all clocks $x \in Y$, $\nu|_Y$ assigns to them the same value as ν. For all clocks $x \in X \setminus Y$, the value of x under $\nu|_Y$ is not defined.

For a valuation ν and a clock constraint φ (as defined in definition 2.4.1),

$$\nu \models \varphi$$

means ν *satisfies* φ. This is defined inductively, using definitions 2.1.9 and 2.1.10 on page 12. □

In addition to the above definition, two operations on valuations are needed.

Definition 2.4.6 (Timeshift, Modification)
Let X and $Y \subseteq X$ sets of clocks, let $\nu : X \to \texttt{Time}$ be a valuation, let $x \in X$ be a clock and let $t \in \texttt{Time}$.

The *timeshift* operation, denoted by $\nu + t$, increases all clocks simultaneously by a given amount of time:

$$(\nu + t)(x) = \nu(x) + t$$

for all clocks x. The intended meaning is: "time t elapses".

The *modification* operation, denoted by $\nu[Y := t]$, resets the values of a given set of clocks and leaves the others unchanged:

$$\nu[Y := t](x) = \begin{cases} t, & x \in Y \\ \nu(x), & \text{otherwise} \end{cases}$$

The intended meaning is: "all clocks $x \in Y$ are reset to t, while all clocks $x \notin Y$ keep their value." □

Using definitions 2.4.5 and 2.4.6, an important property of invariants can be proved: Invariants are convex under timeshift operation (see also (Alur & Dill, 1994)). Roughly speaking, this means: Whenever two valuations ν and $\nu + t$ satisfy the invariant of a certain state s, then all valuations "in between"—that means valuations $\nu + t'$ with $t' \leq t$—also satisfy the invariant. This will now be shown formally.

Lemma 2.4.7 (Convex invariants)
Let $\mathcal{A} = (\underline{A}, S, s_0, X, I, E)$ be a timed automaton, let $t \in \texttt{Time}$, let $\nu, \nu+t \in Val_X$ be valuations, let $s \in S$ be a state, and let $I(s)$ be the invariant of s defined according to definition 2.4.1. Then $I(s)$ is *convex under timeshift operation*, that means

$$(\nu \models I(s)) \wedge (\nu + t \models I(s)) \Rightarrow \forall t' \in \texttt{Time}, t' \leq t : (\nu + t' \models I(s)) \qquad (2.1)$$

Proof. Let $x, y \in X$ be clocks, $\sim \in \{<, \leq, =, \geq, >\}$ and c be a rational constant. Consider a time interval of length t, and let $(\nu \models I(s))$ and $(\nu + t \models I(s))$ (that means the invariant is satisfied at the beginning and at the end of the time interval). The proof is done inductively on the structure of $I(s)$.

IA: $I(s) = x \sim c$:

$(\nu \models I(s))$ and $(\nu + t \models I(s))$ by precondition, that means $\nu(x) \sim c$ and $(\nu+t)(x) \sim c$ by definition of valuation (definition 2.4.5). Thus, $(\nu+t')(x) \sim c$ for all $t' \leq t$ by definition of timeshift and the arithmetical operators[6], and hence $(\nu + t' \models I(s))$.

$I(s) = x - y \sim c$:

[6]Consider for example the arithmetical operator "$\geq$". For $t, t' \in \texttt{Time}, t > t'$: If $\nu(x) \leq c$, and $(\nu + t)(x) \leq c$ for $t \geq 0$, then also $(\nu + t')(x) \leq c$. That means: If the value of clock x is less or equal to c at some time, and time t later still is less or equal to c, than the value of x is less or equal to c for all points in time in between, that means for all $t' < t$.

$(\nu \models I(s))$ and $(\nu + t \models I(s))$ by precondition, that means $(\nu(x) - \nu(y) \sim c)$ and $((\nu+t)(x)-(\nu+t)(y) \sim c)$ by definition of valuation. Thus, $((\nu+t')(x)-(\nu+t')(y) \sim c)$ by definition of timeshift (the value of the difference does not change, as the clocks are increased simultaneously), and hence $(\nu+t' \models I(s))$.

IH: For two invariants φ_1 and φ_2 as defined by definition 2.4.1, let $(\nu + t' \models \varphi_1)$ and $(\nu + t' \models \varphi_2)$.

IS: $I(s) = \varphi_1 \wedge \varphi_2$:

$(\nu \models I(s))$ and $(\nu+t \models I(s))$ by precondition, that means $(\nu \models \varphi_1)$, $(\nu \models \varphi_2)$, $(\nu + t \models \varphi_1)$ and $(\nu + t \models \varphi_2)$ by definition of valuation. Thus, $(\nu + t' \models \varphi_1)$ and $(\nu+t' \models \varphi_2)$ by definition of timeshift and IH, and also $(\nu+t' \models \varphi_1 \wedge \varphi_2)$ by definition of $\wedge$. Hence, $(\nu + t' \models I(s))$.

□

Obviously, allowing invariants to be built with any other formula rules (from definition 2.1.2) than those mentioned in definition 2.4.1 destroys convexity, consider the following example:

Example 2.4.8 (Non-convex clock constraints)
For $\mathcal{A} = (\underline{A}, S, s_0, X, I, E)$ being a timed automaton and $x, y \in X$, consider the "clock constraints"

$$\varphi_1 = (x \leq 5) \vee (y \geq 7)$$
$$\varphi_2 = \neg((x > 5) \wedge (y < 7))$$

and the valuations ν and ν', with $\nu(x) = 4$, $\nu(y) = 4$ and $\nu' = \nu + 4$, that means $\nu'(x) = 8$ and $\nu'(y) = 8$.

Evidently, ν and ν' satisfy φ_1 and φ_2, but for $t \in \texttt{Time}, t = 2$, $\nu'' = \nu + t$ does not satisfy φ_1 and φ_2. Thus φ_1 and φ_2 are not convex in terms of lemma 2.4.7. □

Furthermore, in (Alur & Dill, 1994) and (Alur, 1999), the authors show that for most generalisations of clock constraints (for example allowing more than two clocks in an atomic formula), reachability of states—as used within the properties to be checked here, see section 1.1—is no longer decidable.

Zeno behaviour describes the fact that an infinite sequence of times $t_0, t_1, t_2, \ldots$, $t_i \in \texttt{Time}$, converges to an upper bound $n < \infty$. As real time surely is unbounded, zeno behaviour has to be avoided for timed automata, therefor, the concept of a *real-time sequence* is needed.

Definition 2.4.9 (Real-time Sequence)
An infinite sequence

$$(t_0, t_1, t_2, t_3, \ldots)$$

of values $t_i \in \texttt{Time}$ is called *real-time sequence* iff it satisfies

1. monotony: for all $i \geq 0 : t_i < t_{i+1}$
2. unboundedness: for all $t \in \texttt{Time}$, there exists an $i \geq 1 : t_i > t$

The second property is also called *non-zenoness*. □

According to (Alur, 1999), the operational semantics of a timed automaton is defined by associating a transition system to it.

Definition 2.4.10 (Labelled Transition System (LTS))
A *(labelled) transition system* $\mathcal{S}$ is a tuple $\mathcal{S} = (Q, q_0, \underline{A}, \rightarrow)$ with

- Q is a set of states,
- $q_0 \in Q$ is the initial state,
- $\underline{A}$ is a finite alphabet, the set "labels", and
- $\rightarrow \subseteq Q \times \underline{A} \times Q$ is the set of transitions.

□

The operational semantics of a timed automaton is now given by the following (labelled) transition system.

Definition 2.4.11 (Semantics of a Timed Automaton)
Let $\mathcal{A} = (\underline{A}, S, s_0, X, I, E)$ be a timed automaton. The *associated (labelled) transition system* $\mathcal{S}_\mathcal{A}$ is $\mathcal{S}_\mathcal{A} \stackrel{\text{def}}{=} (Q_\mathcal{A}, (s_0, \nu_0), \texttt{Time} \cup \underline{A}_{?!}, \rightarrow)$ where

- $Q_\mathcal{A} \subseteq S \times Val_X$ is the set of *configurations* of $\mathcal{A}$. A configuration is a pair (s, ν) of a state s of $\mathcal{A}$ and a valuation ν, such that ν satisfies the invariant $I(s)$ of s, that means $\nu \models I(s)$;
- $(s_0, \nu_0) \in Q_\mathcal{A}$ is the initial configuration (also called initial state), with $\nu_0(x) = 0$ for all clocks x;
- there are two types of transitions in $\mathcal{S}_\mathcal{A}$, modelling two different "situations" in $\mathcal{A}$
  - Elapse of time: Represents the automaton remaining in some state while time elapses. For $t \in \texttt{Time}$, $t > 0$, $(s, \nu) \in Q_\mathcal{A}$:
    $$(s, \nu) \xrightarrow{t} (s, \nu + t)$$
    iff $\nu + t' \models I(s)$ for all $0 \leq t' \leq t$. Note that the restriction to values > 0 does not constrain the possible behaviour of the automaton but is used to prevent "useless" transitions of the form $(s, \nu) \xrightarrow{0} (s, \nu) \xrightarrow{0} (s, \nu) \ldots$ as well as for ease of representation, see definition 3.3.21 on page 54.
    A transition of this form is called *delay transition* or *delay step with delay t*.
  - Change of state: Represents a state change in the automaton. For $(s, \nu) \in Q_\mathcal{A}$ and $(s, \tau, \varphi, Y, s') \in E$ such that $\nu \models \varphi$:
    $$(s, \nu) \xrightarrow{\tau} (s', \nu[Y := 0])$$
    A transition of this form is called *action transition (with action τ)*[7].

[7]Remember that visible actions are used for synchronisation of automata only. For this reason, action transitions of a single automaton are only defined for the internal action τ.

A *run of* $\mathcal{S}_\mathcal{A}$ (or *trace in* $\mathcal{A}$) is an infinite sequence of configurations $(s_i, \nu_i) \in Q_\mathcal{A}$ starting in (s_0, ν_0):

$$(s_0, \nu_0) \xrightarrow{\alpha_1} (s_1, \nu_1) \xrightarrow{\alpha_2} (s_2, \nu_2) \xrightarrow{\alpha_3} (s_3, \nu_3) \xrightarrow{\alpha_4} \dots$$

with

- $\forall i \geq 1 : \alpha_i \in \underline{A}_{?!} \cup \texttt{Time}$, that means either a delay step or an action transition is taken
- Let $(\alpha_{i_1}, \alpha_{i_2}, \alpha_{i_3}, \dots)$ be the maximal subsequence of $(\alpha_1, \alpha_2, \alpha_3, \dots)$ with $\alpha_{i_j} \in \texttt{Time}$. The sequence $(\alpha_{i_1}, \alpha_{i_1} + \alpha_{i_2}, \alpha_{i_1} + \alpha_{i_2} + \alpha_{i_3}, \dots)$ is a real-time sequence

According to the definition of labelled transition system, elements in $\alpha \in \underline{A}_{?!} \cup \texttt{Time}$ are called *labels*. For a trace t, the sequence of labels $(\alpha_1, \alpha_2, \alpha_3, \dots)$ is also called the *observable behaviour*.

The *semantics of automaton* $\mathcal{A}$ is given by the set of all possible traces of $\mathcal{S}_\mathcal{A}$, denoted by $Trace_\mathcal{A}$. For $\mathcal{TA}$ being the set of all timed automata (see definition 2.4.3), $Trace_\mathcal{TA}$ denotes the set of all traces for arbitrary automata.

If only finite prefixes are used, let

$$Trace_{\mathcal{A},k} = \{r_k \mid \exists r \in Trace_\mathcal{A}, r_k \text{ is prefix of } r \text{ of length } k\} \text{ and}$$
$$Trace_{\mathcal{TA},k} = \{r_k \mid \exists r \in Trace_\mathcal{TA}, r_k \text{ is prefix of } r \text{ of length } k\}$$

be the set of finite prefixes of length k of traces in $Trace_\mathcal{A}$ and $Trace_{\mathcal{TA},k}$, respectively. Elements in $Trace_{\mathcal{A},k} \cup Trace_{\mathcal{TA},k}$ are called *finite traces* or *traces of length* k. □

Note that in (Alur & Dill, 1994) and (Alur, 1999), the definition of semantics for a single timed automaton allows action transitions labelled with a visible action. In contrast to that, although definition 2.4.3 allows transitions labelled with visible actions, these transitions do not contribute to the semantics of a single automaton (as a transition of the timed automaton used to define an action transition of the associated labelled transition system must be labelled with the internal action τ). Transitions labelled with visible actions are only needed when systems of timed automata are defined (see section 2.4.3).

From definitions 2.4.7 and 2.4.11 follows:

Remark 2.4.12 (Time-additivity)
For two consecutive delay steps in a trace, *time-additivity* holds, that means if $(s, \nu) \xrightarrow{t_1} (s, \nu_1)$ and $(s, \nu_1) \xrightarrow{t_2} (s, \nu_2)$, then also $(s, \nu) \xrightarrow{t_1 + t_2} (s, \nu_2)$, for $t_1, t_2 \in \texttt{Time}$. □

Note that regarding reachability, timed automata with invariants are semantically equivalent (using the semantics defined in definition 2.4.11) to timed automata without invariants. The intuitive idea to proof this is to shift the invariant of some state s to ingoing and outgoing transitions of s.

Lemma 2.4.13 (Equivalence of timed automata with and without invariants)

Let $\mathcal{A} = (\underline{A}, S, s_0, X, I, E)$ be a timed automaton with invariants, let $\nu_0 \models I(s_0)$, for $\nu_0(x) = 0$ for all clocks $x \in X$ and $I(s_0)$ being the invariant of s_0[8]. There exists a timed automaton $\mathcal{B}_\mathcal{A} = (\underline{A}, S, s_0, X, \top, E_\mathcal{B})$, with

$$\top : S \to \{\texttt{true}\}$$

which has the same semantics and does not contain invariants.

Proof. For every state, its invariant will be shifted to in- and outgoing transitions, that means clock constraints will be added to these. In this way, the invariant has to hold at the time where the automaton enters the state, and at the time where the automaton leaves it. Due to lemma 2.4.7 (convex invariants), the invariant will hold in between, too.

For a state with invariant φ and an ingoing transition e, auxiliary function in describes the clock constraint which has to be added to the guard of e. Attention has to be paid to the set of clocks reset by e, as the values of these clocks would be 0 at the time the automaton enters the state. Therefor, in substitutes every occurrence of a reset clock with 0.

Let $e = (s, \alpha, \varphi, \{y_1, \ldots, y_n\}, s')$ be a transition, let $\Theta = \{y_1/0, \ldots, y_n/0\}$ be a substitution, let φ be an invariant. in is given by

$$in(\varphi, e) = \varphi\Theta$$

Note that $in(\varphi, e) = \varphi$ in case φ does not contain clocks which are reset by e, that means $\Sigma(\varphi) \cap Y = \emptyset$. Furthermore, $\Sigma(\varphi\Theta) \cap Y = \emptyset$ by definition of substitution (definition 2.1.7).

For every state $s \in S$, let $I(s)$ be the invariant of s. Using this, $E_\mathcal{B}$ can be defined as

$$E_\mathcal{B} = \left\{ e_\mathcal{B} \;\middle|\; \begin{array}{l} \text{there exists a transition } e = (s, \alpha, \varphi', Y, s') \in E \text{ such that} \\ e_\mathcal{B} = (s, \alpha, (\varphi' \wedge in(I(s'), e) \wedge I(s)), Y, s') \end{array} \right\} \tag{2.2}$$

The trace semantics of $\mathcal{A}$ and $\mathcal{B}_\mathcal{A}$, as defined in definition 2.4.11, is identical, that means

$$Trace_\mathcal{A} = Trace_{\mathcal{B}_\mathcal{A}}.$$

In other words, every trace in $\mathcal{A}$ also is a trace in $\mathcal{B}_\mathcal{A}$, and every trace in $\mathcal{B}_\mathcal{A}$ also is a trace in $\mathcal{A}$[9]:

- $Trace_\mathcal{A} \subseteq Trace_{\mathcal{B}_\mathcal{A}}$: Let t be a trace in $\mathcal{A}$,

  $$t = (s_0, \nu_0) \xrightarrow{\alpha_1} (s_1, \nu_1) \xrightarrow{\alpha_2} (s_2, \nu_2) \xrightarrow{\alpha_3} (s_3, \nu_3) \xrightarrow{\alpha_4} \ldots$$

  t is also a trace in $\mathcal{B}_\mathcal{A}$, as every single step $(s_i, \nu_i) \xrightarrow{\alpha_{i+1}} (s_{i+1}, \nu_{i+1})$ is well-defined with respect to the transition relation of $\mathcal{B}_\mathcal{A}$:

[8] In other words: The invariant for the initial state has to hold if all clocks are set to zero, that means for the initial configuration ν_0

[9] Equivalence can also be shown for labelled transition systems associated with $\mathcal{A}$ and $\mathcal{B}_\mathcal{A}$, respectively, which is a stronger equivalence than trace equivalence. But as the results regarding abstraction are based on traces (and not on transition systems), trace equivalence shall be enough here.

- If $\alpha_{i+1} = t \in \texttt{Time}$, then $\nu_{i+1} = \nu_i + t$ (by definition of delay step), and trivially $\nu_i + t \models \top(s_{i+1})$ as required by definition of delay step (definition 2.4.11)
- If $\alpha_{i+1} \in \underline{A_{?!}}$, there exists a transition $e \in E$, $e = (s_i, \alpha_{i+1}, \varphi, Y, s_{i+1})$, such that $\nu_i \models \varphi$ and $\nu_{i+1} = \nu_i[Y := 0]$ (by definition of action transition). Hence, by definition of $E_{\mathcal{B}}$, there exists a transition $e_{\mathcal{B}} \in E_{\mathcal{B}}$, $e_{\mathcal{B}} = (s_i, \alpha_{i+1}, (\varphi \wedge in(I(s_{i+1}), e) \wedge I(s_i)), Y, s_{i+1})$. ν_i satisfies the guard of $e_{\mathcal{B}}$ because
  - $\nu_i \models \varphi$ by precondition
  - $\nu_i \models I(s_i)$ by definition of configuration (otherwise, t would not have been well-defined)
  - $\nu_i \models in(I(s_{i+1}), e)$: $\nu_{i+1} \models I(s_{i+1})$ by definition of configuration, that means $\nu_i[Y := 0] \models I(s_{i+1})$ (by definition of action transition). Obviously, $\nu_i[Y := 0](x) = 0$ for all clocks $x \in Y$, and therefor $\nu_i[Y := 0] \models I(s_{i+1})\Theta$. $\nu_i[Y := 0] \models in(I(s_{i+1}), e)$ by definition of in, and finally $\nu_i \models in(I(s_{i+1}), e)$ (as $\Sigma(in(I(s_{i+1}), e)) \cap Y = \emptyset$ by definition of substitution).

  Hence, by definition of $\wedge$, $\nu_i \models (\varphi \wedge in(I(s_{i+1}), e) \wedge I(s_i))$, that means ν_i satisfies the guard of $e_{\mathcal{B}}$ as required by definition of action transition.

- $Trace_{\mathcal{A}} \supseteq Trace_{\mathcal{B}_{\mathcal{A}}}$: Let t be a trace in $\mathcal{B}_{\mathcal{A}}$,

$$t = (s_0, \nu_0) \xrightarrow{\alpha_1} (s_1, \nu_1) \xrightarrow{\alpha_2} (s_2, \nu_2) \xrightarrow{\alpha_3} (s_3, \nu_3) \xrightarrow{\alpha_4} \dots$$

  Without loss of generality, let t be *minimal with respect to time-additivity* (see also remark 2.4.12), that means for all $i \geq 0$:

$$(\alpha_i \in \texttt{Time}) \Rightarrow (\alpha_{i+1} \notin \texttt{Time})$$

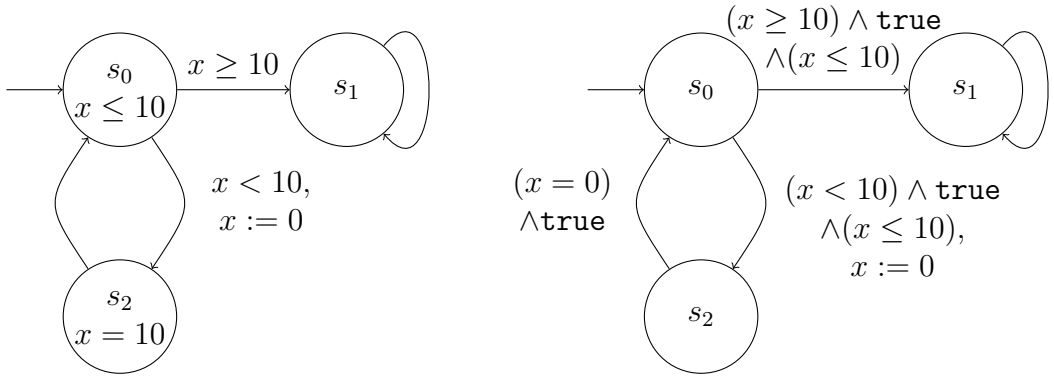
  In other words, t does not contain two consecutive delay steps. t is also a trace in $\mathcal{A}$, as every single step $(s_i, \nu_i) \xrightarrow{\alpha_{i+1}} (s_{i+1}, \nu_{i+1})$ is well-defined with respect to the transition relation of $\mathcal{A}$:

  - If $\alpha_{i+1} \in \underline{A_{?!}}$, there exists a transition $e_{\mathcal{B}} \in E_{\mathcal{B}}$, $e_{\mathcal{B}} = (s_i, \alpha_{i+1}, \varphi_{\mathcal{B}}, Y, s_{i+1})$, such that $\nu_i \models \varphi_{\mathcal{B}}$ and $\nu_{i+1} = \nu_i[Y := 0]$ (by definition of action transition). Hence, by definition of $E_{\mathcal{B}}$, there exists a transition $e \in E$, $e = (s_i, \alpha_{i+1}, \varphi, Y, s_{i+1})$ such that $\varphi_{\mathcal{B}} = (\varphi \wedge in(I(s_{i+1}), e) \wedge I(s_i))$, that means especially $\models \varphi_{\mathcal{B}} \rightarrow \varphi$ (as φ is one conjunctive element of $\varphi_{\mathcal{B}}$). Thus, trivially $\nu_i \models \varphi$ as required by definition of action transition (definition 2.4.11).
  - If $\alpha_{i+1} = t \in \texttt{Time}$, then $\alpha_{i+2} \notin \texttt{Time}$ by precondition, $\nu_{i+1} = \nu_i + t$ by definition of delay step, and the action transition $(s_{i+1}, \nu_{i+1}) \xrightarrow{\alpha_{i+2}} (s_{i+2}, \nu_{i+2})$ is well-defined in $\mathcal{B}_{\mathcal{A}}$ (as trace t is well-defined in $\mathcal{B}_{\mathcal{A}}$). Thus, by definition of $E_{\mathcal{B}}$, there exist transitions $e_{\mathcal{B}} \in E_{\mathcal{B}}$, $e \in E$, $e_{\mathcal{B}} = (s_{i+1}, \alpha_{i+2}, \varphi_{\mathcal{B}}, Y, s_{i+2})$, $e = (s_{i+1}, \alpha_{i+2}, \varphi, Y, s_{i+2})$, such that $\varphi_{\mathcal{B}} = (\varphi \wedge in(I(s_{i+2}), e) \wedge I(s_{i+1}))$ and $\nu_{i+1} \models \varphi_{\mathcal{B}}$. Thus, as $I(s_{i+1})$ is one conjunctive element of $\varphi_{\mathcal{B}}$, $\nu_{i+1} \models I(s_{i+1})$ as required by definition of delay step (definition 2.4.11).

□

Example 2.4.14 (Timed automaton without invariants)
Consider the timed automaton depicted in figure 2.2(a) representing a watchdog. State s_2 has to be visited at least every 10 seconds, and afterwards has to be left immediately. If more than 10 seconds elapse, the transition from s_0 to s_2 is no longer enabled and the automaton has to take the transition to state s_1.

Figure 2.2(b) represents the same automaton after the invariants have been removed according to the above translation (lemma 2.4.13). Note that for explanatory purposes, the guards have not been simplified in figure 2.2(b).



(a) Timed Automaton with Invariants

(b) Timed Automaton without Invariants

Figure 2.2.: Timed Automaton with and without Invariants

□

2.4.3. System of Timed Automata

Timed automata may be combined to form a system of automata. Such systems may be used to realise design decisions, for example, such as distributed systems. The intended meaning of such a system is that the automata contained in the system are working in parallel, while synchronizing via transitions labelled with visible actions.

To be able to synchronize, one automaton has to be able to fire a transition labelled with an input action $a?$, while the other automaton has to be able to fire a transition labelled with an output action $a!$. Furthermore, $a?$ and $a!$ have to be actions of the same underlying channel. One automaton may also fire a transition labelled with the internal action τ, which is independant from other automata. Within the system, either two automata synchronize or one automaton performs an internal action, that means at most two transitions are fired at the same time.

The following definition combines two timed automata to form a system.

Definition 2.4.15 (Coproduct of Timed Automata)
Let $\mathcal{A}_1 = (\underline{A}_1, S_1, s_{0_1}, X_1, I_1, E_1)$ and $\mathcal{A}_2 = (\underline{A}_2, S_2, s_{0_2}, X_2, I_2, E_2)$ be two timed

automata according to definition 2.4.3, with $X_1 \cap X_2 = \emptyset$. The *coproduct of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$, denoted by $\mathcal{A}_1 \| \mathcal{A}_2$, is a new timed automaton $\mathcal{A}_1 \| \mathcal{A}_2 = (\underline{A}, S, s_0, X, I, E)$, with

- $\underline{A} = \underline{A}_1 \cup \underline{A}_2$,
- $S = S_1 \times S_2$,
- $s_0 = (s_{0_1}, s_{0_2})$,
- $X = X_1 \cup X_2$,
- $I(s_1, s_2) = I_1(s_1) \wedge I_2(s_2)$, for $s_1 \in S_1$ and $s_2 \in S_2$,
- The transition relation E is defined as follows:

  For $e_1 = (s_1, \alpha_1, \varphi_1, Y_1, \widetilde{s_1}) \in E_1$ and $e_2 = (s_2, \alpha_2, \varphi_2, Y_2, \widetilde{s_2}) \in E_2$ being a transition of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, such that α_1 and α_2 are input and output action of the same underlying channel,

  $$((s_1, s_2), \tau, Y_1 \cup Y_2, (\widetilde{s_1}, \widetilde{s_2})) \in E. \tag{2.3}$$

  For $e_1 = (s_1, \alpha_1, \varphi_1, Y_1, \widetilde{s_1}) \in E_1$ being a transition of $\mathcal{A}_1$ and $s_2 \in S_2$ being a state in $\mathcal{A}_2$,

  $$((s_1, s_2), \alpha_1, \varphi_1, Y_1, (\widetilde{s_1}, s_2)) \in E. \tag{2.4}$$

  For $e_2 = (s_2, \alpha_2, \varphi_2, Y_2, \widetilde{s_2}) \in E_2$ being a transition of $\mathcal{A}_2$ and $s_1 \in S_1$ being a state in $\mathcal{A}_1$,

  $$((s_1, s_2), \alpha_2, \varphi_2, Y_2, (s_1, \widetilde{s_2})) \in E. \tag{2.5}$$

□

As the coproduct of two timed automata is given as one automaton, the semantics of the coproduct is defined using definition 2.4.11. The intended behaviour of the system (transitions labelled with a visible action must not fire separately) is guaranteed: All traces in $Trace_{\mathcal{A}_1}$ and $Trace_{\mathcal{A}_2}$ are also contained in $Trace_{\mathcal{A}_1 \| \mathcal{A}_2}$ (with the appropriate extension of states). In more detail: Let

$$t_1 = (s_0, \nu_0) \xrightarrow{\alpha_1} (s_1, \nu_1) \xrightarrow{\alpha_2} (s_2, \nu_2) \xrightarrow{\alpha_3} (s_3, \nu_3) \xrightarrow{\alpha_4} \ldots \in Trace_{\mathcal{A}_1}$$

be a trace in $\mathcal{A}_1$, that means $\{s_0, s_1, s_2, s_3\} \subseteq S_1$, $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4\} \subseteq \underline{A}_1$. There exists a trace

$$t_1' = ((s_0, \widetilde{s}), \nu_0) \xrightarrow{\alpha_1} ((s_1, \widetilde{s}), \nu_1) \xrightarrow{\alpha_2} ((s_2, \widetilde{s}), \nu_2) \xrightarrow{\alpha_3} ((s_3, \widetilde{s}), \nu_3) \xrightarrow{\alpha_4} \ldots \in Trace_{\mathcal{A}_1 \| \mathcal{A}_2} \tag{2.6}$$

in $\mathcal{A}_1 \| \mathcal{A}_2$ with the same observable behaviour. This follows directly from (2.4). Equivalently, for a trace $t_2 \in Trace_{\mathcal{A}_2}$, there exists a trace $t_2' \in Trace_{\mathcal{A}_1 \| \mathcal{A}_2}$ with

the same observable behaviour, which follows directly from (2.5). In this way, the functionalities of $\mathcal{A}_1$ and $\mathcal{A}_2$ are combined.

Furthermore, $\mathcal{A}_1$ and $\mathcal{A}_2$ may synchronise via visible actions. Consider a finite prefix of a trace in $\mathcal{A}_1 \| \mathcal{A}_2$,

$$t_k = ((s_0, \widetilde{s_0}), \nu_0) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_3} ((s_k, \widetilde{s_k}), \nu_k) \in Trace_{\mathcal{A}_1 \| \mathcal{A}_2, k}.$$

Let $e_1 = (s_k, \alpha_k, \varphi_k, Y_k, s_{k+1}) \in E_1$ and $e_2 = (\widetilde{s_k}, \widetilde{\alpha_k}, \widetilde{\varphi_k}, \widetilde{Y_k}, \widetilde{s_{k+1}}) \in E_2$ be a transition in $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, such that α_k and $\widetilde{\alpha_k}$ are input and output action of the same underlying channel. Using this,

$$t_{k+1} = ((s_0, \widetilde{s_0}), \nu_0) \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_3} ((s_k, \widetilde{s_k}), \nu_k) \xrightarrow{\tau} ((s_{k+1}, \widetilde{s_{k+1}}) \in Trace_{\mathcal{A}_1 \| \mathcal{A}_2, k}$$

also is a finite trace in $\mathcal{A}_1 \| \mathcal{A}_2$ (this follows directly from (2.3)). This argumentation inductively holds for infinite traces, too.

As this example shows, every two transitions labelled with an input and output action of the same underlying channel may synchronise. It they do so, they must not synchronise with any other transition, therefor, the action of the new transition is τ.

As the coproduct is defined as a single automaton, the semantics of this "system" is given by definition 2.4.11. The restriction that is ensured by the semantics.

To obtain a system of more than two timed automata, definition 2.4.15 is applied several timed to finally obtain one automaton. Surely, a coproduct defined in this way quickly becomes very large. Nevertheless, when defining the formula representation of a system of timed automata, the coproduct is not needed to be defined explicitely. Instead, the intended behaviour is implicitly ensured. Therefor, the inefficient coproduct construction is not needed in practice but only provides the theoretical background. For further details, please refer to section 3.5.

2.5. Uppaal

Uppaal (uppaal, 2005) is an integrated tool environment for modelling, simulation and verification of real-time systems, modelled as networks of timed automata. It is divided in three parts, each of them performing one of the aforementioned tasks. Even though Uppaal thus is a complete model checker according to section 1.1, it does not provide support for abstraction refinement, and therefor great efforts would be necessary to integrate it into an abstraction refinement framework. Nevertheless, for modelling of timed automata, Uppaal provides a simple and intuitive editor, which is intended to be used within the context of this thesis (see section 1.3).

The automaton model defined in Uppaal widely agrees with the one defined in section 2.4. In addition, Uppaal offers some more functionalities which are an extension of definitions 2.4.3 and 2.4.11, as well as of the "classical automaton model" presented in (Alur & Dill, 1994) and (Alur, 1999). These functionalities are for example broadcast channels (used for synchronisation of more than two automata) or committed locations (the location has to be left immediately, that means no delay steps are possible). Furthermore, it is possible to define and use integer variables and arrays.

As these possibilities are an extension of the classical automaton model, they will not be handled any further within the context of this thesis. Note that nevertheless, these additional functionalities have to be catched by the representer whenever UPPAAL is used to generate an input for the SAATRE tool, see section 1.3. That means: The representer has to ensure input files for SAATRE are only constructed from automata corresponding to the SAATRE automaton model. However, it is possible to expand the definition of formula representation of chapter 3 to allow the additional functionalities if desired.

With regard to invariants, UPPAAL restricts the possible clock constraints, compared to those defined in definition 2.4.1, as it does not allow lower bounds on clocks. This limitation only comes into play in case UPPAAL is used to simulate or verify the timed automaton, and therefor does not effect the work of this thesis. As mentioned above, it will be the task of the representer to assure only valid input files are generated for SAATRE .

2.6. Bounded Model Checking for Timed Automata

As mentioned above in section 1.3, only safety properties of the form "a certain state s is not reachable" will be discussed here. Such properties can be expressed using the CTL[10] (Computation Tree Logic) formula $\neg\mathbf{EF}s$, meaning "there exists no path where eventually s holds". Although this is the correct translation of the property, it will be used in its negated form, that means $\mathbf{EF}s$.

This is done for the following reason: Interpolants containing information about the cause why an abstract counterexample is spurious may be derived in case the (unrolled) formula representation of the concrete automaton together with the property is unsatisfiable (see section 1.3). Therefor, the combination of the formulae has to be unsatisfiable in case the state is not reachable. In other words: A satisfying assignment to the unrolled formula representation of the concrete automaton and the property especially satisfies the property. Thus, when working with the property in the form $\mathbf{EF}s$, a satisfying assignment represents a counterexample.

As already mentioned in section 1.1, bounded model checking for timed automata is done by translating the automaton and the property to formulae and then unrolling it up to a fixed depth. Unrolling thus means to instantiate the formula representation of the timed automaton successively for all steps up to the given bound k. In other words: The explicite representation of components of the timed automaton is copied k times to obtain an explicite representation for the first k steps.

For explanatory purposes, when talking about a certain step in this explicite representation, without further defining which step exactly, the following notation is used: For $\mathcal{V}$ being the set of variables representing the automaton *before* the execution step, the set $\mathcal{V}'$ denotes the variables *after* the execution step, where $\mathcal{V}'$ contains a primed variable v' for every variable v in $\mathcal{V}$.

For further considerations on correctness of the formula representation, please note

[10]The usage of temporal logic within the context of this thesis is restricted to this single definition of properties. Therefor, the complete definition of syntax and semantics is omitted, and the reader is referred to (Clarke & Emerson, 1982) instead.

the following: For safety properties, the diameter of the state space of the timed automaton determines the maximal bound for which the existence of counterexamples has to be checked. If this bound is reached and no counterexample is found, the safety property holds. Although it is not always possible to compute the diameter, the main result is that there *exists* a *finite* upper bound, such that in case bounded model checking is applied up to this bound, the result is assured to be correct.

2.6.1. Handling of Events

In general, events are used for synchronisation within a system of timed automata. As long as the "system" consists of only one automaton, all transitions are labelled with the internal action τ, as there is no possibility (and in fact no need) to synchronise. Note that in case a transition of a single automaton is labelled with an event $\alpha \neq \tau$, this transition can never be fired, as there is no possibility to synchronise, and the transition could just as well have been omitted. Thus, for a single automaton, events have no effect on the property to be checked, as they have no influence on reachability of states[11]. For bounded model checking applied to more than one automaton, mainly two synchronisation strategies are possible:

1. Transitions labelled with two corresponding—that means one input and one output—events cannot fire separately but only at the same time.

   As typical use case, consider the very simple mutual exclusion example in figure 2.3. The automata must not enter states c_1 and c_2 ("critical sections") at the same time. This is guaranteed be the synchronisation over event s: As soon as one automaton enters the critical section, the other automaton leaves it.

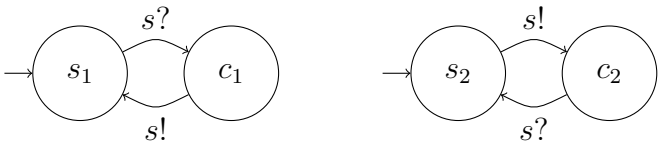


Figure 2.3.: A simple mutual exclusion example

   The mutual exclusion condition can only be ensured if both automata must not fire the transitions on their own but just simultaneously.

   This strategy is mainly used for modelling safety critical systems.

2. Transitions labelled with the same event may fire at the same time or separately.

   As typical use case, consider two communicating processes $p1$ and $p2$, as depicted in figure 2.4 on the next page. Both processes work on some task they

[11] Reachability in a single automaton—apart from the fact that surely there has to be a path from the initial state to the state to be checked—is only restricted by clock valuations, that means invariants and guards.

have to solve together ($work_1$ and $work_2$). The first process send status information about the task, the second one receives those messages (realised through the transitions labelled with sr? and sr!) and acts according to them.

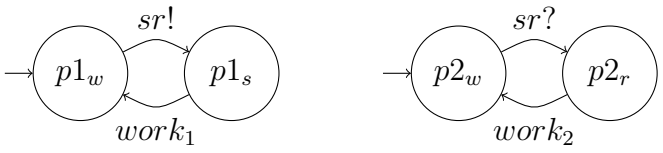


Figure 2.4.: A simple communication example

Imagine the processes are located on different devices, for example in a network. As long as both processes are ready to work, they shall only fire the transition labelled with sr? and sr! together. But if the device containing process two breaks down, process one shall still be able to continue working, perhaps not even knowing his messages will not be noticed by process two any more.

This strategy is used mainly to ensure fault-tolerance/in fault-tolerant systems.

The advantages and disadvantages of the strategies clearly depend on the context they are used in, thus, the choice depends on the systems intended to be represented. As the properties used in this thesis deal with reachability, the first strategy is used (note that UPPAAL also uses the first one)

Remark 2.6.1 (Appearance of events)
In general, no matter which synchronisation strategy for automata is chosen, at most two transitions may fire at the same time. Conceptually, this strategy equals the idea of only one event occurring at the same time. □

2.7. Saatre Overview

To explain the further structure of the thesis, consider figure 2.5, which is an extended version of figure 1.1 on page 5. Grey boxes and arrows have the same meaning as for figure 1.1. In addition, figure 2.5 contains labels denoting the data flow between the components (which has already been described briefly in section 1.3).

From UPPAAL, a timed automaton (TA) and a property (q) are obtained. The Representer converts these to the internal formula format used in SAATRE for representation of timed automata and properties ($\varphi(TA)$ and $\varphi(q)$, respectively). This representation will be defined in chapter 3, as well as the unrolling performed by the Unroller.

The Abstracteur works in the internal formula representation obtained from the Representer and applies some abstraction function α to them, denoted by $\alpha(\varphi(TA))$ and $\alpha(\varphi(q))$, respectively. This is going to be presented in chapter 4. The unrolled abstract formula representation then is given to the DP component (remember that FOCI is used here at the moment).

In case the formula is not satisfiable, this equals to the abstract formula representation being model for the property (or, equivalently: not being model for the negation of the property, see section 2.6), denoted by $\alpha\varphi(\text{TA}) \not\models_k \mathbf{EF}s$. If instead the formula is satisfiable, the DP component will return a trace through the abstract automaton representing a counterexample to the property.

The Concretizer translates the trace back into the notation of the concrete automaton by "undoing" the abstraction. Conceptually, this is done by applying the inverse of abstraction function α to the trace, denoted by $\bar{\alpha}^{1}$ (αtrace/n). Concretization will be dealt with in section 6.2.

Now the concrete formula representation of the automaton is unrolled again and given to FOCI. Note that the depth to unrol the concrete automaton to is not necessarily the same as to unrol the abstract automaton to.

In case the formula is satisfiable, the property is not fulfilled for unrolling depth n. In case the formula is not satisfiable (that means the counterexample is spurious), FOCI will produce interpolants containing information about the possible cause of the unsatisfiability. These are given to the Refiner, which refines the abstraction. Conceptually, the Refiner identifies a set of parameters $\mathcal{P}$ which has to be refined, and the abstraction is done again for the old list of parameters without those parameters contained in $\mathcal{P}$.

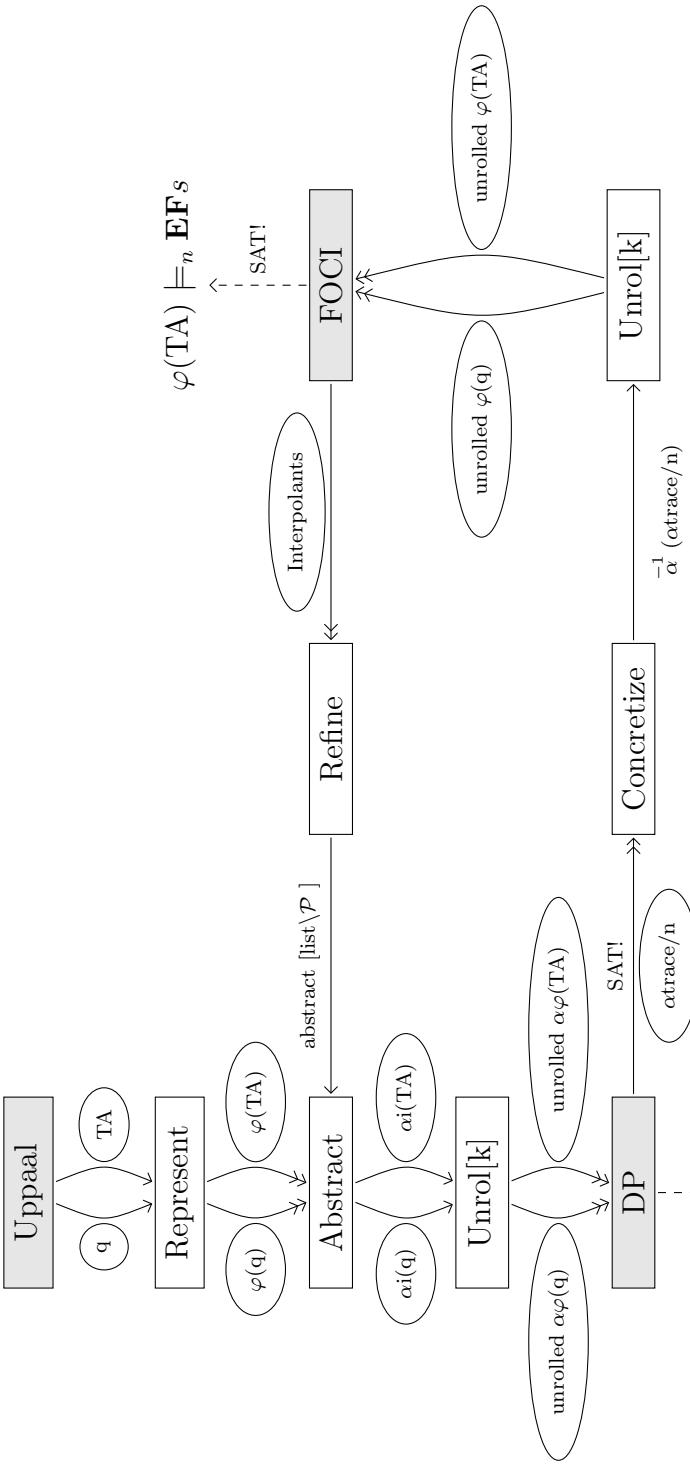


Figure 2.5.: System Architecture for SAT-based Abstraction Refinement Loop, Data flow

3. Representation

When representing timed automata as formulae, many choices of encoding and formula structure are possible, where each of them has different advantages and disadvantages. Although the representation shall be plattform independant, restrictions triggered by the solver intended to be used have to be taken into account. That means, if, for example, the solver is not able to deal with rational variables but only understands integers, the translation has to be aware of this limitation.

In the following section, possible encodings for the most common variable types Boolean, integer, rational, real and enumeration type[1] are presented. In fact, there are many other possibilities of encoding, but most of them are too farfetched and inefficient. Advantages and disadvantages of the encodings are discussed, and examples are shown for every encoding. The examples are intended to provide an insight into the possibilities of encoding and will be presented without formal proof of semantical equivalence (this will be done in section 3.3 when introducing the representation. Section 3.2 sums up the preceding section and motivates the representation which has been choosen within the context of this thesis, provided that the solver is able to deal with variables (and constants) of Boolean and rational type (which is true for FOCI, see section 2.3.2).

Sections 3.3 to 3.5 present the formula representation of a single automaton, a property and a system of automata, respectively. The translations presented are straigt-forward and contain several redundancies. These may be eliminated by using static single assignment form, for example. For further details, see (Cytron *et al.*, 1991), for example.

3.1. Encoding of Variables

Simply speaking, to translate automata to formulae and be able to work with them, all components—as defined in definition 2.4.3 on page 19—have to be encoded with variables. Thus, the main question is which variable type to choose for the representation of the different components. Of course, the most common way of encoding a variable is to simply use a variable with the same type. But apart from the fact that not all SAT solver understand all variable types, a different encoding is often more readable and—most of all—more efficient.

[1] A variable of enumeration type is a variables with a finite range and possibly arbitrary values, like for example {red, green, yellow} or $\{i \in \mathbb{N} \mid i \text{ is prime number and } i \leq 100\}$.

3.1.1. Boolean Variables

It is possible to encode a Boolean variable b with a variable v of integer or enumeration type: Choosing enumeration type, the range of v can be restricted by definition to $\{0, 1\}$, like

```
enum Bool {0, 1} v;
```

Choosing integer, the range has to be restricted "manually" with additional constraints, like for example

```
int v;
0 <= v <= 1;.
```

If the SAT solver allows logical operators within formulae, it is also possible to use a variable with rational or real type:

```
float v;
(v = 0) ∨ (v = 1);
```

No matter which encoding for Boolean variables is chosen, the basic logical operations $\vee, \wedge, \neg$ have to be translated into adequate arithmetical operations modelling their behaviour. Thus, encoding Boolean variables other than with Boolean variables is quite circumstantial due to the required additional constraints and translation of logical operators. Moreover, in most implementations and architectures, using Boolean variables is more efficient than using other variables, so if they not using them will lead to great efficiency loss. Of course, this has to be verified as the case arises.

3.1.2. Enumeration Variables

Encoding of an enumeration variable e with range $D = \{e_1, e_2, \ldots, e_j\}$ may be realised either with one variable v of integer, rational or real type, or with j variables $b_1, b_2, \ldots, b_j$ of Boolean type: Choosing integer, rational or real, the range has to be restricted like above, with an appropriate mapping of the enumeration range to the range of v, for example

```
//Range of e is {red, green, yellow}.
 //Let 1 be red, let 2 be yellow, let 3 be green
          int v;
          1 <= v <= 3;
```

for integer encoding or

```
float v;
(v = 1) ∨ (v = 2) ∨ (v = 3);
```

for rational and real encoding.

Obviously, with increasing number of possible values for e, choosing rational or real variables requires more constraints than choosing consecutive integers.

When choosing Boolean encoding, every variable b_i represents one possible enumeration value e_i, with the intended meaning

$$e = e_i \text{ iff } b_i = \texttt{tt}.$$

The value of e surely is unique at any time. To guarantee the encoding preserves this property, one additional constraint is needed to ensure mutual exclusion of the variables b_i, that means to ensure exactly one variable b_i holds at any time. Using the same example as above, the Boolean encoding is

```
Bool red;
Bool green;
Bool yellow;
(red ∧ ¬yellow ∧ ¬green)∨
(¬red ∧ yellow ∧ ¬green)∨
(¬red ∧ ¬yellow ∧ green).
```

Choosing the integer, rational or real encoding requires not only a completely arbitrary and non-transparent mapping of enumeration values to numerical values, but—as above—additional constraints for range restriction are needed. Restrictions on the possible operations on the variables are needed for all encodings. For example, an operation like `v := v * 5` in the integer encoding example above should not be possible, as it is not defined on the underlying encoding. Although Boolean encoding also requires an additional constraint for mutual exclusion, which is quadratic in the number of possible values for e, this is easier to realise.

3.1.3. Integer Variables

An integer variable i may be encoded with a variable v of rational or real type. No value mapping (as for Boolean and enumeration variables) is required, as the set of integer numbers is a proper subset of the set of rational numbers as well as of the set of real numbers. Attention has to be paid on the possible values v may range over. In most cases considered within this thesis, they will be restricted "automatically"—for example by the transition relation formulae describing how the value of a variable may change. But sometimes they have to be restricted manually, for example like

```
float v;
v DIV 1 = v;
```

where `DIV` denotes the integer division[2].

Although encoding an integer variable with rational or real variables is "possible" in the above sense, most likely it will not be necessary: In the majority of cases,

[2] A division in which the fractional part is discarded is called *integer division*. For example: 10 `DIV` $3 = 3$.

SAT solver able to handle rational or real variables are also able to handle integer variables. Furthermore, in most architectures, time and space complexity of integer variables is less or at most equal, compared to rationals. Thus, it is more efficient to use integers, particularly as no translation is required then.

3.1.4. Rational Variables

The encoding of a rational variable r may be done with a pair (v_n, v_d) of integer variable: Choosing integer, v_n encodes the numerator, v_d encodes the denominator of a fraction representing r (thus using the definition of rational numbers as fractions of two integer numbers). Value mapping is not required for this encoding, but $v_d \neq 0$ has to be ensured for the fraction v_n/v_d to be well-defined. Translating the operations on r to operations on v_n and v_d is quite easy, as the definition arises directly from the corresponding operations in fractional arithmetic. For example,

$$\texttt{x = v + w}$$
$$\texttt{x = v * w}$$

(addition and multiplication on rationals), with `x`, `v` and `w` being rational variables encoded by ($\texttt{x}_n$, $\texttt{x}_d$), ($\texttt{v}_n$, $\texttt{v}_d$) and ($\texttt{w}_n$, $\texttt{w}_d$), respectively, translate to

$$\begin{aligned}\texttt{x}_n &= \texttt{v}_n * \texttt{w}_d + \texttt{v}_d * \texttt{w}_n \\ \texttt{x}_d &= \texttt{v}_d * \texttt{w}_d\end{aligned}$$

for addition and to

$$\begin{aligned}\texttt{x}_n &= \texttt{v}_n * \texttt{w}_n \\ \texttt{x}_d &= \texttt{v}_d * \texttt{w}_d\end{aligned}$$

for multiplication.

Problems in using this encoding arise from the fact that one rational variable may be represented in many ways, for example

$$0.5 = \frac{3}{6} = \frac{2}{4} = \frac{1}{2}.$$

Although the fractional arithmetic operations may be translated to operations on numerators and denominators without requiring a common denominator (and thus avoiding unnecessary reduce/expand operations), most likely the absolute value of numerator and denominator will grow without reducing the fraction from time to time. The main question thus is when to reduce the fraction, as determining whether a fraction is irreducible or not might not be trivial.

3.1.5. Real Variables

Finally, it is possible to encode a real variable with a variable of rational type. Initially, this seems impossible, as the set of real numbers is uncountable, while the set of rational numbers is countable. Nevertheless, without (severe) information loss,

it is possible within the context of this thesis, where clocks are the only variables of real type and appear only in linear constraints (that means linear (in)equalities using the arithmetical operators $\{<, \leq, =, \geq, >\}$, see definitions 2.4.1 and 2.4.3). Furthermore, all constants appearing in the constraints are of rational type.

By using Fourier-Motzkin transformation (FM), (Dantzig & Eaves, 1973) showed that systems of conjoined linear inequalities using only rational constants are satisfiable for rational numbers iff they are satisfiable for real numbers[3]. Although FM transformation is not directly applicable for clock constraints as obtained from definition 2.4.1 (due to the fact that definition 2.4.1 allows all the operators $\vee, \neg, <, \leq, =, \geq, >$), they can be converted into the required format using some simple rewriting rules. This is shown formally using the following lemma.

Lemma 3.1.1 (Rational Encoding for Real Variables)
Let φ be a clock constraint as obtained from definition 2.4.1. φ is *equisatisfiable for rational and real numbers*, that means

$$\varphi \text{ is satisfiable for rational numbers } \Leftrightarrow \varphi \text{ is satisfiable for real numbers} \quad (3.1)$$

Proof. φ is translated into a semantically equivalent formulae $\widetilde{\varphi}$ using the following transformations/steps (all preserving semantics):

1. Transform φ to negation normal form (see appendix A.1.2), this returns φ'
2. Eliminate negation ($\neg$) and the arithmetical operators $=, >$ and $\geq$ from φ' by using function τ, as depicted in figure 3.1. This returns φ''.
3. Transform φ'' to disjunctive normal form (see section appendix A.1.4), this returns $\widetilde{\varphi}$.

Let m be the number of disjunctive elements in $\widetilde{\varphi}$, and let n_i be the number of literals in the i-th disjunctive element, that means

$$\widetilde{\varphi} = (l_{1,1} \wedge l_{1,2} \wedge \ldots \wedge l_{1,n_1}) \vee (l_{2,1} \wedge \ldots \wedge l_{2,n_2}) \vee \ldots \vee (l_{m,1} \wedge \ldots \wedge l_{m,n_1})$$

For every disjunctive element d_i: Divide the set of literals $\{l_{i,1}, \ldots, l_{i,n_i}\}$ into two disjoint sets $\mathcal{L}_<$ and $\mathcal{L}_\leq$, containing all literals with operator $<$ and all literals with operator $\leq$, respectively. By applying Fourier-Motzkin transformation to $\mathcal{L}_<$ and $\mathcal{L}_\leq$ separately:

$$\mathcal{L}_< \text{ is equisatisfiable for rational and real numbers}$$
$$\mathcal{L}_\leq \text{ is equisatisfiable for rational and real numbers}$$

and thus, by definition of satisfiability of clause sets, also

$$\mathcal{L}_< \cup \mathcal{L}_\leq \text{ is equisatisfiable for rational and real numbers.}$$

Every disjunctive element of $\widetilde{\varphi}$ is equisatisfiable for rational and real numbers, hence, by definition of satisfiability and $\vee$, $\widetilde{\varphi}$ is equisatisfiable for rational and real numbers. As φ is semantically equivalent to $\widetilde{\varphi}$, finally 3.1 holds. □

[3] "Satisfiable for rational numbers": There exists a mapping σ_q assigning a rational value to each variable, that means $\sigma(x_i) \in \mathbb{Q}$, such that the system of inequalities is satisfiable. "Satisfiable for real numbers": There exists a mapping σ_r assigning a real value to each variable, that means $\sigma(x_i) \in \mathbb{R}$, such that the system of inequalities is satisfiable

$$
\begin{aligned}
\tau(x < c) &= (x < c) & \tau(x - y < c) &= (x - y < c)\\
\tau(x \leq c) &= (x \leq c) & \tau(x - y \leq c) &= (x - y \leq c)\\
\tau(x = c) &= (x \leq c) \wedge & \tau(x - y = c) &= (x - y \leq c) \wedge\\
&\quad (-x \leq -c) & &\quad (y - x \leq -c)\\
\tau(x \geq c) &= (-x \leq -c) & \tau(x - y \geq c) &= (y - x \leq -c)\\
\tau(x > c) &= (-x < -c) & \tau(x - y > c) &= (y - x < -c)\\
\tau(\neg(x < c)) &= (-x \leq -c) & \tau(\neg(x - y < c)) &= (x - y \geq c)\\
\tau(\neg(x \leq c)) &= (-x < -c) & \tau(\neg(x - y \leq c)) &= (y - x < -c)\\
\tau(\neg(x = c)) &= (-x < -c) \vee & \tau(\neg(x - y = c)) &= (x - y < c) \vee\\
&\quad (x < c) & &\quad (y - x < -c)\\
\tau(\neg(x \geq c)) &= (x < c) & \tau(\neg(x - y \geq c)) &= (x - y < c)\\
\tau(\neg(x > c)) &= (x \leq c) & \tau(\neg(x - y > c)) &= (x - y \leq c)
\end{aligned}
$$

$\tau(\varphi)$ eliminates negation ($\neg$) and the operators $=, \geq, >$ from a clock constraint φ. Without loss of generality, assume φ is in negation normal form (see appendix A.1.2), note that $\tau(\varphi)$ also is in negation normal form. τ preserves the semantics, as it uses only well-defined arithmetical equalities.

Figure 3.1.: Reduction to operator set $\{\wedge, \vee, =, \leq, <\}$ by function τ.

3.1.6. Conclusion

Whenever a variable v with a certain type is encoded using a variable e—or several variables $e_1, \ldots, e_n$—with a different type, additional constraints restricting the possible values of e and $e_1, \ldots, e_n$, respectively, are needed. Thus, the SAT solver (or any other tool intended to use the encoded variables) has to offer the operators needed for these constraints. For example, if rational encoding for integer variables is used, the solver has to be able to axiomatise the effects of integer division. The choice of encoding thus depends—at least partly—on the functionalities provided by the solver. But as these functionalities are not always known at the time the encoding is defined, and the encoding shall be platform-independent, it is often preferable to choose an encoding using only common or basic functionalities (like for example Boolean operations), although this may not always be the most efficient encoding. A main task in the choice of an appropriate encoding thus is to find a trade-off between universality and efficiency, which is also platform-independent.

Regarding efficiency, Boolean encoding is supposed to be more efficient than the other possibilities. Within the context of this thesis, different encodings have been evaluated, and heuristic tests provided support for the assumption. Furthermore—as the name implies—Boolean operators should be present in every SAT solver. No matter which encoding is chosen, attention has to be paid whenever a variable of a certain type is encoded using a variable of another type: Dependent on the platform intended to use the encoding (and the exact choice of encoding), efficiency may greatly vary.

3.2. Characteristics of an Adequate Representation

In the following, the choice of encoding for the different components of timed automata (as defined in definition 2.4.3 on page 19) is explained, provided that the solver is able to deal with variables and constants of Boolean and rational type (which is true for FOCI). This section shall help to understand why a certain encoding has been chosen, and therefor concentrates on considerations regarding efficiency and universality. The exact formula representation will be explained in detail in section 3.3. It is inspired by the encoding presented in (Audemard *et al.*, 2002), but slightly different. The differences will be explained in the following, too.

Both the set of states and the set of events are encoded using Boolean variables, one variable per state and event, respectively (linear Boolean encoding). In (Audemard *et al.*, 2002), the encoding of events is realised in the same way, while logarithmic Boolean encoding is used for states[4]. The main reason for preferring Boolean variables (to integer variables, for example) is a great difference in efficiency: Heuristic tests showed that Boolean encoding is at least 20 times faster than rational encoding, and FOCI is able to deal with much larger formulae in case these (mainly) consist of Boolean variables.

Of course, a bottleneck of the Boolean encoding is the additional mutual exclusion constraint. This constraint is needed for states to ensure exactly one state variable holds at any time. Otherwise, as there exists one variable per state, the automaton would be allowed to be in more than one state at the same time. For events, the constraint is required to guarantee the synchronisation strategy (see section 2.6.1) is preserved. As the size of the mutual exclusion constraint is quadratic in the number of Boolean variables, it may slow down the solver with an increasing number of states and events, respectively. Nevertheless, the efficiency loss in case of rational encoding mainly turned the balance.

Although the mutual exclusion constraint would not have been needed in case of logarithmic Boolean encoding, linear Boolean encoding has been chosen. This is due to the fact that in the representation of (Audemard *et al.*, 2002), every state variable appears twice in every transition. In the encoding presented here, the mutual exclusion constraint is needed only once, and transitions will only contain two state variables. Thus, with a decreasing number of states and an increasing number of transitions, the linear Boolean encoding is more efficient. Furthermore, the mutual exclusion constraint will not be needed in case the solver is able to deal with linear zero-one constraints (see for example (Fränzle & Herde, 2005)).

In (Audemard *et al.*, 2002), clocks are encoded by means of real variables, while in the representation presented in section 3.3, rational variables are used. As shown in lemma 3.1.1, within the context of timed automata used in this thesis, rational encoding is sufficient for real-valued variables, that means no necessary information is lost. Thus, for the representation of clocks, rational variables have been chosen.

For the encoding of transitions, several alternatives regarding the structure of the formula representing a transition (and the change of values it causes) have been

[4]Logarithmic Boolean encoding means: For $\{s_1, \ldots, s_n\}$ being the set of states, n being the number of states, $\lceil \log_2(n) \rceil$ variables are introduced, representing a $\lceil \log_2(n) \rceil$-digit Boolean number s. The intended meaning is that the value of s is i iff the system is in state s_i.

shortlisted. For explanatory purposes, let e be a transition from state s_1 to state s_2, let φ be a formula containing all variables representing precondition of e (for example clocks), and let φ' be a formula containing all variables representing the postcondition of e. The exact structure of φ and φ' is of no importance here, as they are only used to reveal the conceptual idea of transition encoding. The three main alternatives of encoding a single transition from s_1 to s_2 are

$$s_1 \wedge \varphi \rightarrow s_2 \wedge \varphi' \tag{3.2}$$

$$s_1 \wedge \varphi \leftarrow s_2 \wedge \varphi \tag{3.3}$$

$$s_1 \wedge \varphi \wedge s_2 \wedge \varphi' \tag{3.4}$$

To obtain the complete transition relation, first every transition is translated on its own. The complete transition relation is obtained from the conjunction of these single formulae in case of (3.2) and (3.3) and from the disjunction of the single formulae in case of (3.4).

Remember properties to be checked within the context of this thesis are safety properties of the form "a certain state s can never be reached", see also section 2.6. With regard to verification of such properties, the encoding used in (3.2) reflects the idea of starting in the initial state and, while following the transition relation in forward direction, trying to reach the undesired state s. In contrast to that, the translation presented in (3.3) reflects the idea of starting in the undesired state s and, while following the transition relation in backward direction, trying to reach initial state. Finally, the third alternative depicted in (3.4) reflects the idea that firing a transition is only possible if the preconditions (φ) as well as the postconditions (φ') hold.

The worst case with respect to efficiency is given in case the formula is not satisfiable, where the SAT solver has to try all possible valuations. Thus, fixed valuations help in improving the efficiency (as the solver does not have to test these). Comparing (3.2) and (3.3), "more" valuations are known in the former case, as not only the values of the state variables are given, but also the initial valuations for all clocks are known. Hence, (3.2) will be more efficient than (3.3).

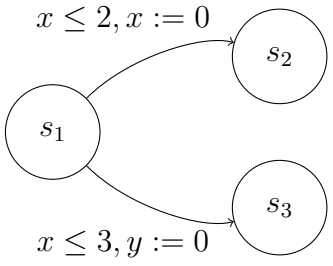


Figure 3.2.: Mutual exclusion of states example

Remember a primed version v' of variable v denotes the value of v after a transition has fired, see section 2.6. For comparison of (3.2) and (3.4), consider the automaton extract shown in figure 3.2. Choosing the encoding of (3.2), the transition relation

would be translated like

$$(s_1 \wedge (x \leq 2) \rightarrow s_2' \wedge (x' = 0) \wedge (y' = y)) \wedge$$
$$(s_1 \wedge (x \leq 3) \rightarrow s_3' \wedge (y' = 0) \wedge (x' = x)),$$

while choosing the encoding of (3.2), it would be translated like

$$(s_1 \wedge (x \leq 2) \wedge s_2' \wedge (x' = 0) \wedge (y' = y)) \vee$$
$$(s_1 \wedge (x \leq 3) \wedge s_3' \wedge (y' = 0) \wedge (x' = x)).$$

(Note that this example does *not* represent the final encoding, but is only used to make clear advantages and disadvantages of the different encoding strategies.)

The non-exclusive guards lead to nondeterminism in case the value of clock x is less or equal to 2, $(x \leq 2)$. When using the encoding of (3.2), this has to be handled explicitely, as the premises of both implications are satisfied, but due to mutual exclusion of states not both conclusion can be satisfied at the same time. In the latter case (that means when using the encoding of (3.4)), the nondeterminism is preserved, as it is not clear which formula will be fulfilled by the SAT solver.

The encoding of transitions presented in (Audemard *et al.*, 2002) mainly works the same (apart from the exact encoding of states, see above), but the authors introduce a new Boolean variable T_e for every transition e, with the intended meaning that T_e holds iff the timed automaton fires transition e. For $\widetilde{e}$ being the formula representation of transition e, a single transition is encoded using the implication $T_e \rightarrow \widetilde{e}$ in (Audemard *et al.*, 2002), while it is encoded using only formula $\widetilde{e}$ in this thesis.

Furthermore, two more Boolean variables T_δ and T_{null} are introduced in (Audemard *et al.*, 2002), with the intended meaning that T_δ holds iff the timed automaton does a delay step with delay δ (no matter which state the automaton is in), and T_{null} holds if the timed automaton "does nothing", that means no variable changes its value. Within the context of this thesis, delay steps are encoded explicitely for every state s, as they have to contain constraints for preserving the invariants.

3.3. Formula Representation of a Single Automaton

The translation is presented in several steps: In section 3.3.1, the "basic" components of timed automata (clocks, clock constraints and states)—as defined in definition 2.4.3 on page 19—are translated. Afterwards, in section 3.3.2, these translations are combined to receive translations for the "complex" components (transition relation and additional constraints), and section 3.3.4 proves the correctness of this representation. Sections 3.4 and 3.5 present the representation of properties and systems of automata, respectively

The translation of a single automaton is presented without consideration of events, as events $\neq \tau$ are used for synchronisation between automata only, and a single automaton contains only the internal action. Thus, when working with a single automaton, all action transitions are labelled with τ, so τ is not needed to distinguish them. Events will be handled in section 3.5, too.

The next sections will define the formula representation of a timed automaton $\mathcal{A}$ syntactically, but the concept behind the representation only becomes clear with regard to an interpretation of the variables introduced to encode $\mathcal{A}$. Some components will therefor be explained using an interpretation σ. Please consider remark 3.3.12 for further details.

Remark 3.3.1 (Notation of Timed Automata)
In the next sections, let $\mathcal{A} = (\underline{A}, S, s_0, X, I, E)$ be a timed automaton, with $\underline{A} = \{\alpha_1, \ldots, \alpha_n\}$, $X = \{x_0, \ldots, x_m\}$ and $S = \{s_0, \ldots, s_n\}$. As long as only one automaton is considered, the internal action τ will be omitted in the transitions, as explained above. A transition $(s, \tau, \varphi, Y, s') \in E$ therefor may be abbreviated with (s, φ, Y, s'). □

3.3.1. Basic Components

Definition 3.3.2 (Representation of States)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1. For every state $s \in S$, a timed Boolean variable $\overline{\mathtt{s}_t}$ is introduced, with the intended meaning

$$\sigma(\overline{\mathtt{s}_t}) = \mathtt{tt} \;\Leftrightarrow\; \text{at step } t\text{, the automaton is in state } s \tag{3.5}$$

for some interpretation σ of the variables introduced to encode $\mathcal{A}$. $\overline{\mathtt{s}}_t$ is called *encoding of* s. □

As mentioned above, clocks are modelled by means of rational-valued (or simply rational) variables, see also lemma 3.1.1.

Definition 3.3.3 (Representation of Clocks)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1. A new clock $z \notin X$—called *absolute time reference*—is introduced, measuring the time that has passed during the run of $\mathcal{A}$. z is represented by a timed rational variable $\overline{\mathtt{z}}_t$, called *encoding of* z. The intended meaning is

$$\sigma(\overline{\mathtt{z}}_t) = c \;\Leftrightarrow\; \text{at step } t\text{, time amount } c \text{ elapsed since the beginning of computation} \tag{3.6}$$

for some interpretation σ of the variables introduced to encode $\mathcal{A}$.

For every clock $x \in X$, a rational timed variable $\overline{\mathtt{x}}_t$ is introduced, with the intended meaning

$$\sigma(\overline{\mathtt{x}}_t) = c \Leftrightarrow c \text{ is the absolute time when } x \text{ was last reset.} \tag{3.7}$$

Let $\overline{\mathtt{X}}$ be the set containing all these variables $\overline{\mathtt{x}}_t$. The value of $\overline{\mathtt{x}}_t$ only changes when some transition in the automaton resets x, and $\overline{\mathtt{z}}$ will never be reset, hence,

$$\sigma(\overline{\mathtt{z}}_t) - \sigma(\overline{\mathtt{x}}_t) = c \;\Leftrightarrow\; \text{the value of clock } x \text{ at step } t \text{ is } c \tag{3.8}$$

The difference $\overline{\mathtt{z}}_t - \overline{\mathtt{x}}_t$ is called *encoding of* x. □

The encoding of clock values using the difference $\sigma(\overline{\mathtt{z}}_t) - \sigma(\overline{\mathtt{x}}_t)$ (instead of simply using $\sigma(\overline{\mathtt{x}}_t)$) has been chosen to be able to specify reachability properties on the one hand, but, which is more important, the absolute time reference will greatly ease the proof of correctness. See section 3.3.4 for this purpose.

The value of a clock x at step t is given by $\overline{\mathtt{z}_t}$ - $\overline{\mathtt{x}_t}$, therefor, this difference is used in clock constraints whenever the value of x is needed.

Definition 3.3.4 (Representation of Clock Constraints)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1. For $x, y \in X$ being clocks, c being a rational constant and $\sim \in \{<, \leq, =, \geq, >\}$, the encoding of the clock constraints $x \sim c$ and $x - y \sim c$ (see definition 2.4.1 on page 18) is given by

$$\begin{aligned} \overline{\mathtt{z}}_t - \overline{\mathtt{x}}_t &\sim c \\ \overline{\mathtt{y}}_t - \overline{\mathtt{x}}_t &\sim c \end{aligned} \tag{3.9}$$

respectively[5]. Clock constraints of the form $\varphi_1 \wedge \varphi_2$ and $\neg\varphi_1$ are inductively defined as the conjunction and negation of the above. □

3.3.2. Complex Components

For a state change (which is instantaneous), the encoding has to ensure the clocks which are not reset keep their values, while those which are reset are set to the absolute point in time. The state variables change according to the states involved in the transition, and the guard has to hold.

Definition 3.3.5 (Representation of an Action Transition)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1. Let $(s, \varphi, Y, s') \in E$, let $\overline{\mathtt{s}}_t$ be the encoding of s and $\overline{\mathtt{s}}'_t$ be the encoding of s', respectively. The *formula representation of action transition* (s, φ, Y, s') is given by

$$\begin{aligned} e_a((s, \varphi, Y, s')) \overset{def}{=} \overline{\mathtt{s}}_t \wedge \overline{\mathtt{s}}'_{t+1} \wedge \varphi_t \wedge (\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1}) \wedge \\ \bigwedge_{x \in Y} (\overline{\mathtt{x}}_{t+1} = \overline{\mathtt{z}}_{t+1}) \wedge \bigwedge_{x \notin Y} (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}), \end{aligned} \tag{3.10}$$

with φ_t being the representation of guard φ at step t, according to definition 3.3.4. A formula obtained from this definition is called an *action transition formula.* □

As this translation has to be done for every action transition, it leads to $|E|$ formulae of the above form.

To model the automaton remaining in one of his states while time elapses, delay transitions have to be added to the encoding (compare definition 2.4.11 on page 23). The formulae have to ensure the state's invariant holds at the beginning and at the end of the time delay[6], neither the state variables nor the variables for the automaton's clocks ($x \in X$) change, but the value of the additional clock z increases by the value of the time delay.

[5] In fact, the encoding is $(\overline{\mathtt{z}}_t - \overline{\mathtt{x}}_t) - (\overline{\mathtt{z}}_t - \overline{\mathtt{y}}_t) \sim c$, but this simplifies to $\overline{\mathtt{y}}_t - \overline{\mathtt{x}}_t \sim c$

[6] By ensuring this, they will hold in between, too, as invariants are convex clock constraints, see lemma 2.4.7

Definition 3.3.6 (Representation of Delay Steps)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1, let $\overline{\mathtt{s}}_t$ be the encoding of state s. The *formula representation for a delay step for s* (that means the automaton remains in state s while time elapses, see definition 2.4.11 on page 23) is given by

$$e_d(s) \stackrel{def}{=} \overline{\mathtt{s}}_t \wedge \overline{\mathtt{s}}_{t+1} \wedge I_t(\overline{\mathtt{s}}_t) \wedge I_{t+1}(\overline{\mathtt{s}}_{t+1}) \wedge \bigwedge_{x \in X} (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{z}}_t < \overline{\mathtt{z}}_{t+1}) \tag{3.11}$$

with $I_t(\overline{\mathtt{s}}_t)$ being the clock constraint representation of the invariant of state $\overline{\mathtt{s}}$ at time t. A formula obtained from this definition is called *delay transition formula.*
□

As this translation has to be done for every state, it leads to $|S|$ formulae of the above form.

The automaton may either fire a transition or take a delay step, hence the complete transition relation of the automaton (representing its behaviour) is obtained by combining action and delay transitions.

Definition 3.3.7 (Representation of the Complete Transition Relation)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1. The *complete transition relation of* $\mathcal{A}$ is given by

$$\varphi_E(\mathcal{A}) \stackrel{def}{=} \bigvee_{(s,\varphi,Y,s') \in E} e_a((s,\varphi,Y,s')) \vee \bigvee_{s \in S} e_d(s) \tag{3.12}$$

Thus, the complete transition relation is the disjunction of action and delay transitions obtained from definitions 3.3.5 and 3.3.6. □

Initial conditions restrict the values of variables at time 0, that means before the beginning of computation. Thus, a formula defining the initial state and the initial values of the clocks (all zero!) is added to the formula set.

Definition 3.3.8 (Representation of Initial Conditions)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1, let s_0 be its initial state and $\overline{\mathtt{s\_0}}_0$ the encoding of the initial state. The *representation of initial conditions* is given by

$$\varphi_i(\mathcal{A}) \stackrel{def}{=} \overline{\mathtt{s\_0}}_0 \wedge (\overline{\mathtt{z}}_0 = 0) \wedge \bigwedge_{x \in X} (\overline{\mathtt{x}}_0 = 0) \tag{3.13}$$

□

Clearly, the automaton always is in only one of his states, which is implicitly satisfied by definition 2.4.11. For the encoding chosen here, the constraint is expressed by the requirement that only one of the state variables may hold at the same time, and has to be ensured "manually" for equation (3.5) to hold. Otherwise, it would be possible to find an interpretation σ with $\sigma(\overline{\mathtt{s}}) = \mathtt{tt}$ and $\sigma(\overline{\mathtt{s}}') = \mathtt{tt}$, for $s, s' \in S, s \neq s'$, representing the automaton being in two of his states at the same time. To prevent such erroneous behaviour, the following mutual exclusion constraint is added to the encoding.

Definition 3.3.9 (Representation of Mutual State Exclusion)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1, and let $\overline{\mathtt{S}} = \{\overline{\mathtt{s\_0}}, \ldots, \overline{\mathtt{s\_n}}\}$ be the set of variables representing the set of states S. The mutual exclusion of states is ensured by the formula

$$\varphi_{gc}(\mathcal{A}) \stackrel{def}{=} \bigvee_{0 \leq i \leq n} (\overline{\mathtt{s\_i}}_t \wedge \bigwedge_{\substack{0 \leq k \leq n \\ k \neq i}} (\neg \overline{\mathtt{s\_k}}_t)). \tag{3.14}$$

Mutual exclusion is not determined by the automaton itself but arises from the encoding. Therefor, this formula is called *general constraint representation.* □

The complete representation of a timed automaton $\mathcal{A}$ is obtained by the conjunction of all formulae as defined above.

Definition 3.3.10 (Representation of Timed Automaton)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1.

The conjunction

$$\varphi(\mathcal{A}) \stackrel{def}{=} \varphi_{gc}(\mathcal{A}) \wedge \varphi_E(\mathcal{A}) \wedge \varphi_i(\mathcal{A}), \tag{3.15}$$

with $\varphi_{gc}(\mathcal{A})$, $\varphi_E(\mathcal{A})$ and $\varphi_i(\mathcal{A})$ obtained from (3.14), (3.12) and (3.13), respectively (thus representing the general constraints and the constraints determined by the automaton) is called *formula representation of* $\mathcal{A}$.

For $\mathcal{TA}$ being the set of all timed automata (see definition 2.4.3 on page 19), let $\varphi(\mathcal{TA})$ be the set of all formula representations. □

3.3.3. Unrolling

The formulae defined above are abstract in that they are defined for some abstract execution step t which is expressed by the use of relative variables $\overline{\mathtt{v}}_t$ and $\overline{\mathtt{v}}_{t+1}$. To be able to define the unrolling of an automaton, for each abstract formula, a concrete formula representing the component at (the concrete) unrolling depth $k \in \mathbb{N}$ is obtained by simply substituting every occurrence of variable $\overline{\mathtt{v}}_t$ with $\overline{\mathtt{v}}_k$ and every occurrence of variable $\overline{\mathtt{v}}_{t+1}$ with $\overline{\mathtt{v}}_{k+1}$.

To unroll the automaton up to depth k the concrete formulae for depth $0, 1, \ldots, k-1, k$ are combined.

Definition 3.3.11 (Concrete Formula Representation, k-unrolling of Timed Automaton)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1, let $k \in \mathbb{N}$, let $\Theta_k = \{\overline{\mathtt{s}}_t/\overline{\mathtt{s}}_k, \overline{\mathtt{x}}_t/\overline{\mathtt{x}}_k, \overline{\mathtt{z}}_t/\overline{\mathtt{z}}_k, \overline{\mathtt{s}}_{t+1}/\overline{\mathtt{s}}_{k+1}, \overline{\mathtt{x}}_{t+1}/\overline{\mathtt{x}}_{k+1}, \overline{\mathtt{z}}_{t+1}/\overline{\mathtt{z}}_{k+1}\}$ be a substitution for all states $s \in S$, all clocks $x \in X$ and the global clock z.

For $e_d(s)$, $e_a((s, \varphi, Y, s'))$, $\varphi_E(\mathcal{A})$ and $\varphi_{gc}(\mathcal{A})$ being the formula representations obtained from definitions 3.3.5, 3.3.6, 3.3.7 and 3.3.9, respectively, the *concrete*

formula representations are given by

$$e_d(s)_k \stackrel{def}{=} e_d(s)\Theta_k \tag{3.16}$$

$$e_a((s,\varphi,Y,s'))_k \stackrel{def}{=} e_a((s,\varphi,Y,s'))\Theta_k \tag{3.17}$$

$$\varphi_E(\mathcal{A})_k \stackrel{def}{=} \bigvee_{(s,\varphi,Y,s')\in E} (e_a((s,\varphi,Y,s'))_k) \vee \bigvee_{s\in S} (e_d(s)_k) \tag{3.18}$$

$$\varphi_{gc}(\mathcal{A})_k \stackrel{def}{=} \varphi_{gc}(\mathcal{A})\Theta_k \tag{3.19}$$

The unrolling of automaton $\mathcal{A}$ up to depth k, $k \geq 1$, called *k-unrolling of $\mathcal{A}$*, can now be defined by

$$\varphi(\mathcal{A})_k \stackrel{def}{=} \bigwedge_{0\leq j\leq k} (\varphi_{gc}(\mathcal{A})_j) \wedge \bigwedge_{0\leq j\leq k-1} (\varphi_E(\mathcal{A})_j) \wedge \varphi_i(\mathcal{A})^7 \tag{3.20}$$

For $\mathcal{TA}$ being the set of all timed automata, let $\varphi(\mathcal{TA})_k$ be the set of all k-unrollings for automata. □

Remark 3.3.12 (Unrolling)
The intended meaning of the k-unrolling of some automaton $\mathcal{A}$ is to represent the complete possible behaviour of $\mathcal{A}$ for the first k steps. An interpretation $\sigma \in \Sigma(\varphi(\mathcal{A})_k)$, with $\sigma \models \varphi(\mathcal{A})_k$, therefor represents one concrete behaviour for the first k steps. Thus, a trace of the automaton is given by a satisfying valuation of the formula. □

Note that for such a finite trace of length k, the formula representation of the transition relation, $\varphi_E(\mathcal{A})$, is only unrolled up to depth $k-1$, as $\varphi_E(\mathcal{A})_j$ represents a (delay or action) step from depth j to depth $j+1$.

Two results directly arising from the above definitions will be presented, as they are going to be used for the abstraction and refinement.

Remark 3.3.13 (Containedness of k-unrollings)
By definition, for all unrolling depths k and k': $\varphi_{gc}(\mathcal{A})_{k'}$ is a subformula of $\varphi_{gc}(\mathcal{A})_k$, that means the k'-unrolling of the mutual exclusion constraint is contained in the k-unrolling, for $k' \leq k$. Equivalently, $\varphi_E(\mathcal{A})_{k'}$ is a subformula of $\varphi_E(\mathcal{A})_k$, that means the k'-unrolling of the transition relation is contained in the k-unrolling, for $k' \leq k$. Thus, apart from commuting, $\varphi(\mathcal{A})_k$ contains $\varphi(\mathcal{A})_{k'}$ as subformula (as $\varphi_i(\mathcal{A})$ does not change for any unrolling depth), that means the unrolled formula for depth k contains the unrolled formulae for all smaller depths $k' \leq k$. □

Remark 3.3.14 (Negation Normal Form)
For a timed automaton $\mathcal{A} = (\underline{A}, S, s_0, X, I, E)$, let $\mathcal{A}^{nnf} = (\underline{A}, S, s_0, X, I^{nnf}, E^{nnf})$ be a timed automaton which is equal to $\mathcal{A}$ apart from the fact that every clock constraint φ (that means either guard or invariant) of $\mathcal{A}$ is transformed to its negation normal form $NNF(\varphi)$ (see appendix A.1.2) in $\mathcal{A}^{nnf}$. As $NNF(\varphi)$ is semantically equivalent to φ, $\mathcal{A}$ and $\mathcal{A}^{nnf}$ are also equivalent (that means they have the associated transition systems and thus the same semantics, see definition 2.4.11). Furthermore, $\varphi(\mathcal{A})^{nnf}$ as well as $\varphi(\mathcal{A}^{nnf})_k$ are in negation normal form.

[7] Surely, $\varphi_i(\mathcal{A})$ is not unrolled, as it has to hold for $k = 0$ only.

Proof. This follows directly form definitions 3.3.7, 3.3.8, 3.3.9, 3.3.10 and 3.3.11. □

Thus, for every timed automaton $\mathcal{A}$, there exists a timed automaton $\mathcal{A}^{nnf}$ with the same semantics, such that the formula representation $\varphi(\mathcal{A})$ and k-unrolling $\varphi(\mathcal{A})_k$ of $\mathcal{A}^{nnf}$ are in negation normal form. Hence, without loss of generality, $\varphi(\mathcal{A})$ and $\varphi(\mathcal{A})_k$ are assumed to be in negation normal form in the sequel.

Example 3.3.15 (Encoding of Timed Automaton)
Consider the automaton $\mathcal{A}$ as depicted in figure 3.3. The automaton has four states (s_0, s_1, s_2, s_3) and three clocks (v, x, y). With the encoding of timed automata as defined above, it is encoded as follows. For readability, the internal action τ as well as guards and invariants equal to `true` will be omitted in the formula representation.

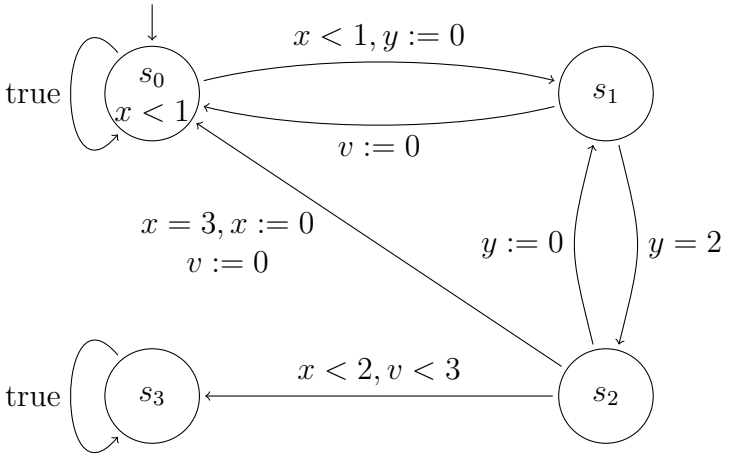


Figure 3.3.: A simple automaton example

From definition 3.3.5 (action transitions), the following formulae are obtained:

$$
\begin{aligned}
&\overline{\mathtt{s0}}_t \wedge \overline{\mathtt{s0}}_{t+1} \wedge (\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1}) \wedge (\overline{\mathtt{v}}_t = \overline{\mathtt{v}}_{t+1}) \wedge (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{y}}_t = \overline{\mathtt{y}}_{t+1})\\
&\overline{\mathtt{s0}}_t \wedge \overline{\mathtt{s1}}_{t+1} \wedge (\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1}) \wedge (\overline{\mathtt{y}}_{t+1} = \overline{\mathtt{z}}_{t+1}) \wedge (\overline{\mathtt{v}}_t = \overline{\mathtt{v}}_{t+1}) \wedge (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1})\\
&\overline{\mathtt{s1}}_t \wedge \overline{\mathtt{s0}}_{t+1} \wedge (\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1}) \wedge (\overline{\mathtt{v}}_{t+1} = \overline{\mathtt{z}}_{t+1}) \wedge (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{y}}_t = \overline{\mathtt{y}}_{t+1})\\
&\overline{\mathtt{s1}}_t \wedge \overline{\mathtt{s2}}_{t+1} \wedge (\overline{\mathtt{z}}_t - \overline{\mathtt{y}}_t = 2) \wedge (\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1})\\
&\qquad\qquad \wedge (\overline{\mathtt{v}}_t = \overline{\mathtt{v}}_{t+1}) \wedge (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{y}}_t = \overline{\mathtt{y}}_{t+1})\\
&\overline{\mathtt{s2}}_t \wedge \overline{\mathtt{s0}}_{t+1} \wedge (\overline{\mathtt{z}}_t - \overline{\mathtt{x}}_t = 3) \wedge (\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1})\\
&\qquad\qquad \wedge (\overline{\mathtt{v}}_{t+1} = \overline{\mathtt{z}}_{t+1}) \wedge (\overline{\mathtt{x}}_{t+1} = \overline{\mathtt{z}}_{t+1}) \wedge (\overline{\mathtt{y}}_t = \overline{\mathtt{y}}_{t+1})\\
&\overline{\mathtt{s2}}_t \wedge \overline{\mathtt{s1}}_{t+1} \wedge (\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1}) \wedge (\overline{\mathtt{y}}_{t+1} = \overline{\mathtt{z}}_{t+1}) \wedge (\overline{\mathtt{v}}_t = \overline{\mathtt{v}}_{t+1}) \wedge (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1})\\
&\overline{\mathtt{s2}}_t \wedge \overline{\mathtt{s3}}_{t+1} \wedge (\overline{\mathtt{z}}_t - \overline{\mathtt{x}}_t < 2) \wedge (\overline{\mathtt{z}}_t - \overline{\mathtt{v}}_t < 3) \wedge (\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1})\\
&\qquad\qquad \wedge (\overline{\mathtt{v}}_t = \overline{\mathtt{v}}_{t+1}) \wedge (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{y}}_t = \overline{\mathtt{y}}_{t+1})
\end{aligned}
$$

From definition 3.3.6 (delay transitions), the following formulae are obtained:

$$\begin{aligned}
&\overline{\mathtt{s0}}_t \wedge \overline{\mathtt{s0}}_{t+1} \wedge (\overline{\mathtt{z}}_t - \overline{\mathtt{x}}_t < 1) \wedge (\overline{\mathtt{z}}_{t+1} - \overline{\mathtt{x}}_{t+1} < 1) \\
&\qquad\qquad \wedge (\overline{\mathtt{v}}_t = \overline{\mathtt{v}}_{t+1}) \wedge (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{y}}_t = \overline{\mathtt{y}}_{t+1}) \wedge \wedge (\overline{\mathtt{z}}_t < \overline{\mathtt{z}}_{t+1}) \\
&\overline{\mathtt{s1}}_t \wedge \overline{\mathtt{s1}}_{t+1} \wedge (\overline{\mathtt{v}}_t = \overline{\mathtt{v}}_{t+1}) \wedge (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{y}}_t = \overline{\mathtt{y}}_{t+1}) \wedge \wedge (\overline{\mathtt{z}}_t < \overline{\mathtt{z}}_{t+1}) \\
&\overline{\mathtt{s2}}_t \wedge \overline{\mathtt{s2}}_{t+1} \wedge (\overline{\mathtt{v}}_t = \overline{\mathtt{v}}_{t+1}) \wedge (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{y}}_t = \overline{\mathtt{y}}_{t+1}) \wedge \wedge (\overline{\mathtt{z}}_t < \overline{\mathtt{z}}_{t+1}) \\
&\overline{\mathtt{s3}}_t \wedge \overline{\mathtt{s3}}_{t+1} \wedge (\overline{\mathtt{v}}_t = \overline{\mathtt{v}}_{t+1}) \wedge (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{y}}_t = \overline{\mathtt{y}}_{t+1}) \wedge \wedge (\overline{\mathtt{z}}_t < \overline{\mathtt{z}}_{t+1})
\end{aligned}$$

The complete transition relation $\varphi_E(\mathcal{A})$ according to definition 3.3.7 is obtained from the disjunction of these eleven formulae.

From definition 3.3.8 (initial conditions), the following formula is obtained:

$$\varphi_i(\mathcal{A}) = \overline{\mathtt{s0}}_0 \wedge (\overline{\mathtt{z}}_0 = 0) \wedge (\overline{\mathtt{v}}_0 = 0) \wedge (\overline{\mathtt{x}}_0 = 0) \wedge (\overline{\mathtt{y}}_0 = 0)$$

From definition 3.3.9 (mutual exclusion), the following formula is obtained:

$$\begin{aligned}
\varphi_{gc}(\mathcal{A}) = &(\overline{\mathtt{s0}}_t \wedge \neg\overline{\mathtt{s1}}_t \wedge \neg\overline{\mathtt{s2}}_t \wedge \neg\overline{\mathtt{s3}}_t) \\
&\vee (\neg\overline{\mathtt{s0}}_t \wedge \overline{\mathtt{s1}}_t \wedge \neg\overline{\mathtt{s2}}_t \wedge \neg\overline{\mathtt{s3}}_t) \\
&\vee (\neg\overline{\mathtt{s0}}_t \wedge \neg\overline{\mathtt{s1}}_t \wedge \overline{\mathtt{s2}}_t \wedge \neg\overline{\mathtt{s3}}_t) \\
&\vee (\neg\overline{\mathtt{s0}}_t \wedge \neg\overline{\mathtt{s1}}_t \wedge \neg\overline{\mathtt{s2}}_t \wedge \overline{\mathtt{s3}}_t)
\end{aligned}$$

The formula representation of $\mathcal{A}$, $\varphi(\mathcal{A})$, is obtained from the conjunction

$$\varphi(\mathcal{A}) = \varphi_{gc}(\mathcal{A}) \wedge \varphi_E(\mathcal{A}) \wedge \varphi_i(\mathcal{A})$$

as defined in definition 3.3.10. □

3.3.4. Correctness Proof

By definition (definition 2.4.11), automaton traces are infinite sequences of configurations. In contrast to that, the transition relations as defined above only represents finite prefixes of traces. Nevertheless, the translation can be used for model checking: According to section 2.6, there exists a finite upper bound up to which bounded model checking has to be applied to the system to ensure correctness of the results. This bound is the diameter of the system. Although it may be hard or even impossible to compute the diameter, the existence is enough for the formula representation to yield correct results. If the system is checked with increasing values of k (the unrolling depth), the diameter will finally be reached, and the results for this unrolling depth and all greater unrolling depths are surely correct.

For a timed automaton $\mathcal{A}$ and its k-unrolling $\varphi(\mathcal{A})_k$ as defined in definition 3.3.11,

define the following formulae:

$$\begin{aligned}\text{MUTEX} &= \bigwedge_{0\le j\le k} (\varphi_{gc}(\mathcal{A})_j) \\ &= \bigwedge_{0\le j\le k} \Big(\bigvee_{0\le i\le n} (\overline{\texttt{s\_i}}_j \wedge \bigwedge_{\substack{0\le k\le n \\ k\neq i}} (\neg \overline{\texttt{s\_k}}_j)) \Big) \end{aligned} \tag{3.21}$$

$$\text{CLOCKS\_INIT} = (\overline{\texttt{z}}_0 = 0) \wedge \bigwedge_{x\in X} (\overline{\texttt{x}}_0 = 0) \tag{3.22}$$

$$\text{CLOCKS} = \bigwedge_{0\le j\le k} (\overline{\texttt{z}}_j \le \overline{\texttt{z}}_{j+1}) \wedge \bigwedge_{x\in X} (\overline{\texttt{z}}_j - \overline{\texttt{x}}_j \ge 0) \wedge \bigwedge_{x\in X} (\overline{\texttt{x}}_j \le \overline{\texttt{x}}_{j+1}) \tag{3.23}$$

$$\text{MCCI} = \text{MUTEX} \wedge \text{CLOCKS} \wedge \text{CLOCKS\_INIT} \tag{3.24}$$

As mentioned in remark 3.3.12, an interpretation σ of the k-unrolling of some automaton $\mathcal{A}$ which satisfies $\varphi(\mathcal{A})_k$ represents a trace of length k. Roughly speaking, MCCI defines the constraints such an interpretation σ has to satisfy at least to result in a well-defined trace: The mutual exclusion constraint surely has to be satisfied, as has been explained in sections 3.2 and 3.3.2. The formula denoted in (3.22) expresses the fact that the initial value of all clocks has to be zero. By (3.23), the correct behaviour of clocks over time is assured: The absolute time reference must not decrease ($\overline{\texttt{z}}_j \le \overline{\texttt{z}}_{j+1}$), clocks do not have negative values ($\overline{\texttt{z}}_j - \overline{\texttt{x}}_j \ge 0$) and—as $\overline{\texttt{x}}_t$ represents the time where x was last reset—the sequence of times where a clock is reset must not decrease, too ($\overline{\texttt{x}}_j \le \overline{\texttt{x}}_{j+1}$). This property will be used in the sequel to show the equivalence of traces and interpretations.

The correctness of the the formula representation is shown by proving figure 3.4 is a commuting diagram. First, a correspondence between finite traces of length k and interpretations of k-unrollings is defined (lower part of figure 3.4). Note that not all interpretations are considered, but only those who are a model for MCCI. Afterwards, all other subparts of figure 3.4 are shown to be a commuting diagram.

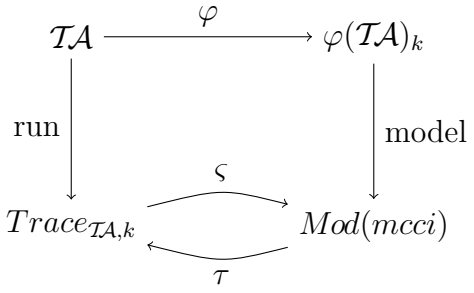


Figure 3.4.: Overview of Formula Representation of Timed Automata, Commuting Diagram

Remark 3.3.16 (Notation of Representation)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1. In the following, let $\overline{\texttt{s\_i}}$ be the variable representing state s_i, let $\overline{\texttt{x}}$ be the representation of clock x_0. Let $\overline{\texttt{X}}$ be the set containing the representations of all clocks in X (see definition 3.3.3), and let $\overline{\texttt{z}}$ be the representation of the absolute time reference z. □

Remark 3.3.17 (Absolute time reference)
A trace t does not necessarily use (that means assign values to) the absolute time reference z, as this is an additional clock used for representation only. Nevertheless, for every trace not using z, there exists an equivalent trace using z: z is added to the set of clocks, and

$$\nu_0(z) = 0$$
$$\nu_{k+1}(z) = \begin{cases} \nu_k(z), & \text{if } (s,\nu_k) \xrightarrow{\tau} (s',\nu_{k+1}) \text{ is an action transition} \\ \nu_k(z)+t, & \text{if } (s,\nu_k) \xrightarrow{t} (s',\nu_{k+1}) \text{ is a delay step with delay } t \end{cases}$$

□

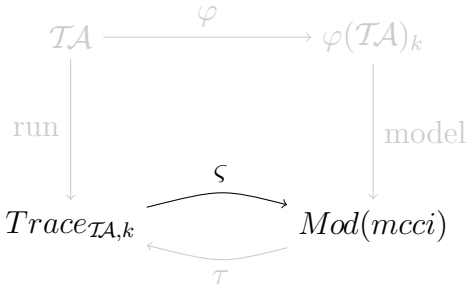


Figure 3.5.: Definition of corresponding interpretation

Definition 3.3.18 (Corresponding interpretation)
Let

$$r_k : (s_0,\nu_0) \xrightarrow{\alpha_0} (s_1,\nu_1) \xrightarrow{\alpha_1} (s_2,\nu_2) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{k-1}} (s_k,\nu_k) \in Trace_{\mathcal{TA},k}$$

be a finite trace of length k. Without loss of generality, assume r_k uses z according to remark 3.3.17.

For $0 \leq k' \leq k$, the *corresponding interpretation* σ_{r_k} (see figure 3.5) is defined as:

$\sigma_{r_k}(\overline{\texttt{s\_j}}_{k'})$	$= \texttt{tt}$,	for $\overline{\texttt{s\_j}}$ representing state $s_j = s_{k'}$
$\sigma_{r_k}(\overline{\texttt{s\_l}}_{k'})$	$= \texttt{ff}$,	for $\overline{\texttt{s\_l}}$ representing state $s_l, s_l \neq s_j$
$\sigma_{r_k}(\overline{\texttt{z}}_0)$	$= 0$	
$\sigma_{r_k}(\overline{\texttt{z}}_{k'})$	$= \nu_{k'}(z)$,	for $\overline{\texttt{z}}$ representing the absolute time reference z
$\sigma_{r_k}(\overline{\texttt{x}}_{k'})$	$= \sigma_{r_k}(\overline{\texttt{z}}_{k'}) - \nu_{k'}(x)$,	for $\overline{\texttt{x}}$ representing clock x

Let $\varsigma : Trace_{\mathcal{TA},k} \to Mod(mcci)$ be the mapping assigning the corresponding interpretation to every trace. □

Remember the absolute time reference $\overline{\texttt{z}}$ shall represent the time passed since the beginning of computation (definition 3.3.3). As r_k uses z according to remark 3.3.17, the value of $\overline{\texttt{z}}$ is directly given by the valuation.

The general concept behind the corresponding interpretation σ_{r_k} for a trace r_k of automaton $\mathcal{A}$ is to read off values from the trace such that the interpretation is a model for the unrolling of an automaton, $\sigma_{r_k} \models \varphi(\mathcal{A}')_k$. At the moment, the automata $\mathcal{A}$ and $\mathcal{A}'$ are not further qualified.

It will be shown now that the corresponding interpretation is well-defined in that ς indeed maps to $Mod(mcci)$

Remark 3.3.19 (Corresponding interpretation is in $Mod(mcci)$)
Let r_k be a finite trace of length k,

$$r_k : (s_0, \nu_0) \xrightarrow{\alpha_0} (s_1, \nu_1) \xrightarrow{\alpha_1} (s_2, \nu_2) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{k-1}} (s_k, \nu_k)$$

let σ_{r_k} be the corresponding interpretation. Then:

$$\sigma_{r_k} \in Mod(mcci)$$

Proof.

1. $\sigma_{r_k} \models$ MUTEX: Let $\{s_0, \ldots, s_n\}$ be the set of state variables appearing in configurations of r_k, let $\{\overline{\mathtt{s\_0}}_{k'}, \ldots, \overline{\mathtt{s\_n}}_{k'}\}$ be the set of representations for the state variables after step k'.

   Consider configuration $(s_{k'}, \nu_{k'})$. For all $k' > 0$, by definition $\sigma_{r_k}(\overline{\mathtt{s\_t}}_{k'}) = \mathtt{tt}$ for state $s_t = s_{k'}$, and $\sigma_{r_k}(\overline{\mathtt{s\_j}}_{k'}) = \mathtt{ff}$ for all other states $s_j \neq s_t$. That means

   $$\sigma_{r_k} \models (\bigwedge_{0 \leq i \leq t-1} (\neg \overline{\mathtt{s\_i}}_{k'}) \wedge \overline{\mathtt{s\_t}}_{k'} \wedge \bigwedge_{t+1 \leq j \leq n} (\neg \overline{\mathtt{s\_j}}_{k'}))$$

   and thus σ_{r_k} is a model for the whole disjunction:

   $$\sigma_{r_k} \models \varphi_{gc}(\mathcal{A})_{k'}$$

   As $\sigma_{r_k} \models \varphi_{gc}(\mathcal{A})_{k'}$ for $0 \leq k' \leq k$, finally

   $$\sigma_{r_k} \models \bigwedge_{0 \leq k' \leq k} \varphi_{gc}(\mathcal{A})_{k'}$$

2. $\sigma_{r_k} \models$ CLOCKS:

   $\sigma_{r_k} \models (\overline{\mathtt{z}}_k \leq \overline{\mathtt{z}}_{k'})$ for all k, k': This follows directly from definition 3.3.18 and remark 3.3.17.

   $\sigma_{r_k} \models (\overline{\mathtt{z}}_k - \overline{\mathtt{x}}_k \geq 0)$ for all k: This follows directly from definition 3.3.18, as $\sigma_{r_k}(\overline{\mathtt{z}}_k) - \sigma_{r_k}(\overline{\mathtt{x}}_k) = \nu_{k'}(z) - (\nu_{k'}(z) - \nu_{k'}(x)) = \nu_{k'}(z)$, and $\nu_{k'}(z) > 0$ according to remark 3.3.17.

   $\sigma_{r_k} \models (\overline{\mathtt{x}}_k \leq \overline{\mathtt{x}}_{k'})$ for all k, k': This follows directly from definition 3.3.18 and definition of configuration and valuation (as either $\nu_k(x) \geq \nu_k(x)$ in case of an action transition which resets x, or $\nu_k(x) \geq \nu_k(x)$ in case of an action transition which does not reset x, or $\nu_k(x)$ and $\nu_k(z)$ increase by the same amount of time in case of a delay transition).

   $$\begin{aligned} & \sigma_{r_k}(\overline{\mathtt{x}}_k) \\ =& \sigma_{r_k}(\overline{\mathtt{z}}_k) - \nu_k(x) \\ =& \sigma_{r_k}(\overline{\mathtt{z}}_k) - \nu_k(x) \\ \leq& \sigma_{r_k}(\overline{\mathtt{z}}_{k'}) - \nu_{k'}(x) \\ =& \sigma_{r_k}(\overline{\mathtt{x}}_{k'}) \end{aligned}$$

3. $\sigma_{r_k} \models$ CLOCKS_INIT: Directly follows from definition 3.3.18, as $\sigma_{r_k}(\overline{\mathtt{z}}_0) = 0$, $\sigma_{r_k}(\overline{\mathtt{x}}_0) = \sigma_{r_k}(\overline{\mathtt{z}}_0) - \nu_0(x)$, and furthermore $\nu_{k'}(x)$ by definition of valuation and configuration.

□

Remark 3.3.20 (Corresponding interpretation satisfies invariants)
Let $\mathcal{A} \in \mathcal{TA}$ be a timed automaton, let r_k be a finite trace of length k in $\mathcal{A}$,

$$r_k : (s_0, \nu_0) \xrightarrow{\alpha_0} (s_1, \nu_1) \xrightarrow{\alpha_1} (s_2, \nu_2) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{k-1}} (s_k, \nu_k)$$

let σ_{r_k} be the corresponding interpretation.

For every configuration $(s_{k'}, \nu_{k'})$, σ_{r_k} satisfies the invariant of $s_{k'}$, that means

$$\sigma_{r_k} \models I_{k'}(\overline{\mathtt{s\_k'}}_{k'}),$$

for $\overline{\mathtt{s\_k'}}_{k'}$ representing state $s_{k'}$.

Proof. This follows directly from definition 3.3.18. □

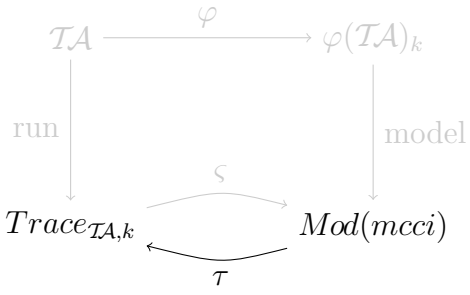


Figure 3.6.: Definition of corresponding trace

Definition 3.3.21 (Corresponding Trace)
Let $\varphi(\mathcal{A})_k \in \varphi(\mathcal{TA})_k$ be the k-unrolling of a timed automaton for some given $k \in \mathbb{N}$, and let $\sigma \in Mod(mcci)$.

The *corresponding trace* t_σ (see figure 3.6) ,

$$t_\sigma : (s_0, \nu_0) \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{k-1}} (s_k, \nu_k),$$

is defined as

$$\begin{array}{lll}
s_{k'} & = s, & \text{for } \overline{\mathtt{s_j}} \text{ representing state } s \text{ with } \sigma(\overline{\mathtt{s_j}}_{k'}) = \mathtt{tt} \quad (*) \\
\nu_{k'}(z) & = \sigma(\overline{\mathtt{z}}_{k'}), & \text{for } \overline{\mathtt{z}} \text{ representing the absolute time reference } z \\
\nu_{k'}(x) & = \sigma(\overline{\mathtt{z}}_{k'}) - \sigma(\overline{\mathtt{x}}_{k'}), & \text{for } \overline{\mathtt{x}} \text{ representing clock } x \\
\alpha_{k'} & = \begin{cases} \tau & \text{for } \sigma(\overline{\mathtt{z}}_{k'}) = \sigma(\overline{\mathtt{z}}_{k'+1}) \\ t & \text{for } t \in \mathtt{Time}, t > 0 \text{ and } \sigma(\overline{\mathtt{z}}_{k'}) + t = \sigma(\overline{\mathtt{z}}_{k'+1}) \end{cases}
\end{array}$$

Let $\tau : Mod(mcci) \to Trace_{\mathcal{TA},k}$ be the mapping assigning the corresponding trace to every interpretation $\sigma \in Mod(mcci)$. □

As this is only a definition, it is not clear that the trace obtained from it is well-defined. To proof the corresponding trace is a proper trace, it has to be shown (*) is satisfied for a single state only (otherwise, s would not be well-defined)

It also has to be shown the time is modelled correctly. That means the initial values are zero, the value of z increases over time, the value of a clock x always is greater or equal zero and for a step from k to $k+1$, either the value of x increases by the same amount as z (otherwise, it would be possible for x to run with a different speed than z) or it is set to zero.

Note it is not necessary to check whether for a configuration, the valuation satisfies the invariant of the state, as according to lemma 2.4.13, the state can be assumed to have no invariant.

Remark 3.3.22 (Corresponding trace is well-defined)
The corresponding trace is well-defined in the sense of definition 2.4.11, that means for all $k' \leq k$:

1. there exists exactly one unique state $s_{k'}$ satisfying condition (*) and

2. time is modelled correctly, that means

   - for the absolute time reference z: $\nu_0(z) = 0$ and $\nu_{k'}(z) \leq \nu_{k'+1}(z)$
   - for all clocks $x \in X$: $\nu_0(x) = 0$, $\nu_{k'}(x) \geq 0$ and $((\nu_{k'+1}(x) - \nu_{k'}(x) = \nu_{k'+1}(z) - \nu_{k'}(z))$ or $(\nu_{k'+1}(x) = 0))$.

Proof. Consider configuration (s_k, ν_k).

1. $\sigma \models$ MUTEX by precondition (as $\sigma \in Mod(mcci)$), that means $\sigma \models \varphi_{gc}(\mathcal{A})_k$:

$$\sigma \models (\bigwedge_{0 \leq i \leq t-1} (\neg \overline{\texttt{s\_i}}_k) \wedge \overline{\texttt{s\_t}}_k \wedge \bigwedge_{t+1 \leq j \leq n} (\neg \overline{\texttt{s\_j}}_k)).$$

   Hence, $\sigma(\overline{\texttt{s\_t}}_k) = \texttt{tt}$ only for state variable $\overline{\texttt{s\_t}}$, and $\sigma(\overline{\texttt{s\_j}}_k) = \texttt{ff}$ for all other state variables $\overline{\texttt{s\_j}}$. Thus, there exists a unique state s (represented by $\overline{\texttt{s\_t}}$) satisfying condition (*).

2. $\sigma \models$ CLOCKS_INIT and $\sigma \models$ CLOCKS by precondition, as $\sigma \in Mod(mcci)$

   - $\sigma(\overline{\texttt{z}}_0) = 0$ and $\sigma(\overline{\texttt{z}}_{k'}) \leq \sigma(\overline{\texttt{z}}_{k'+1})$ for all $k' \leq k$ by precondition. Thus, $\nu_0(z) = 0$ and $\nu_{k'}(z) = \nu_{k'+1}(z)$ by definition of corresponding trace.
   - for all clocks $x \in X$: $\sigma(\overline{\texttt{x}}_0) = 0$, $\sigma(\overline{\texttt{x}}_{k'}) \leq \sigma(\overline{\texttt{x}}_{k'+1})$ and $\sigma(\overline{\texttt{z}}_{k'}) - \sigma(\overline{\texttt{x}}_{k'}) \geq 0$ by precondition.

     Thus, $\nu_0(x) = 0$, $\nu_{k'}(x) \geq 0$ and $((\nu_{k'+1}(x) - \nu_{k'}(x) = \nu_{k'+1}(z) - \nu_{k'}(z))$ or $(\nu_{k'+1}(x) = 0))$ by definition of corresponding trace and action and delay transition.

□

For figure 3.4 to be a commuting diagram, the following properties have to hold:

The results of the model checking have to be correct, that means for every automaton $\mathcal{A}$, its k-unrolling $\varphi(\mathcal{A})_k$ and an interpretation $\sigma \in Mod(mcci)$ being model for the k-unrolling, the interpretation must be the corresponding interpretation for a trace of the automaton, so that the same result can be obtained from a trace in $\mathcal{A}$ (see figure 3.8).

The results of the model checking have to be complete, that means for every automaton $\mathcal{A}$, its k-unrolling $\varphi(\mathcal{A})_k$, an interpretation being model for the k-unrolling and the corresponding trace, this trace really must be a trace of the automaton (see figure 3.9)

These properties will now be shown formally.

Lemma 3.3.23 (Identity of corresponding trace and interpretation)
The composition (see definition 2.1.6) of corresponding trace and corresponding interpretation returns the identity, that means

$$\varsigma \circ \tau = \tau \circ \varsigma = id.$$

See also figure 3.7.

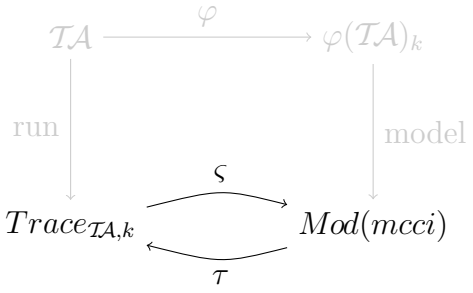


Figure 3.7.: Identity of corresponding trace and interpretation

Proof.

1. Let $\varphi(\mathcal{A})_k \in \varphi(\mathcal{TA})_k$ be the k-unrolling of a timed automaton for some given $k \in \mathbb{N}$, let $\sigma \in Mod(mcci)$, let t_σ be the corresponding trace of σ, let σ'_{t_σ} be the corresponding interpretation of t_σ. Then

   $$\sigma = \sigma'_{t_\sigma}.$$

   Let $\{\overline{\texttt{s\_0}}, \ldots, \overline{\texttt{s\_n}}\}$ be the set of state variables. For every k' with $0 \leq k' \leq k$, let $\{\overline{\texttt{s\_0}}_{k'}, \ldots, \overline{\texttt{s\_n}}_{k'}\} \subset Mod(mcci)$ be the set of representations for states after step k', let $\{s_0, \ldots, s_n\}$ be the set of states in t_σ, and let $(s_{k'}, \nu_{k'})$ be the configuration of t_σ after step k'.

   - $\sigma \models$ MUTEX by precondition, that means there exists exactly one state variable $\overline{\texttt{s\_j}}$ with $\sigma(\overline{\texttt{s\_j}}_{k'}) = \texttt{tt}$. By definition of corresponding trace, $s_j = s_{k'}$ for $\overline{\texttt{s\_j}}$ representing s_j. By definition of corresponding interpretation, $\sigma_{t_\sigma}(\overline{\texttt{s\_j}}) = \texttt{tt}$, and $\sigma_{t_\sigma}(\overline{\texttt{s\_l}}) = \texttt{ff}$ for all states $\overline{\texttt{s\_l}} \neq \overline{\texttt{s\_j}}$. Thus,

     $$\sigma(\overline{\texttt{s}}_{k'}) = \sigma'_{t_\sigma}(\overline{\texttt{s}}_{k'}) \text{ for all states } s$$

- $\sigma \models$ CLOCKS_INIT by precondition, that means $\sigma(\overline{\mathtt{z}}_0) = 0$. For the absolute time reference $\overline{\mathtt{z}}$:

$$\sigma(\overline{\mathtt{z}}_{k'}) = \sigma(\overline{\mathtt{z}}_{k'}) - \sigma(\overline{\mathtt{z}}_0) = \nu_{k'}(\overline{\mathtt{z}}_{k'}) - \nu_0(\overline{\mathtt{z}}_0) = \overset{def.\varsigma}{=} \sigma'_{t_\sigma}(\overline{\mathtt{z}}_{k'})$$

- For all clocks $x \in X$:

$$\sigma(\overline{\mathtt{x}}_{k'}) = \sigma(\overline{\mathtt{z}}_{k'}) - (\sigma(\overline{\mathtt{z}}_{k'}) - \sigma(\overline{\mathtt{x}}_{k'})) \overset{\sigma(\overline{\mathtt{z}}_{k'})=\sigma'_{t_\sigma}(\overline{\mathtt{z}}_{k'})}{=} \sigma'_{t_\sigma}(\overline{\mathtt{z}}_{k'}) - (\sigma(\overline{\mathtt{z}}_{k'}) - \sigma(\overline{\mathtt{x}}_{k'})) \overset{def.\tau}{=} \sigma'_{t_\sigma}(\overline{\mathtt{z}}_{k'}) - \nu_{k'}(x) \overset{def.\varsigma}{=} \sigma'_{t_\sigma}(\overline{\mathtt{x}}_{k'})$$

Thus,

$$\varsigma \circ \tau = id.$$

2. Let

$$t : (s_0, \nu_0) \xrightarrow{\alpha_0} (s_1, \nu_1) \xrightarrow{\alpha_1} (s_2, \nu_2) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{k-1}} (s_k, \nu_k) \in Trace_{\mathcal{TA},k}$$

be a finite trace of length k. Without loss of generality, assume t uses the absolute time reference z (see remark 3.3.17). Let σ_t be the corresponding interpretation of t, let t'_{σ_t} be the corresponding trace of σ_t. Then

$$t = t'_{\sigma_t}.$$

Let $(s_{k'}, \nu_{k'})$ and $(s'_{k'}, \nu'_{k'})$ be the configurations of t and t'_{σ_t}, respectively, after step k', $0 \leq k' \leq k$.

- By definition of ς, $\sigma(\overline{\mathtt{s\_j}}_{k'}) = \mathtt{tt}$ for $\overline{\mathtt{s\_j}}$ representing $s_{k'}$, and $\sigma(\overline{\mathtt{s\_l}}_{k'}) = \mathtt{ff}$ for all other states $s_l \neq s_{k'}$. That means, only the variable representing $s_{k'}$ is set to $\mathtt{tt}$. By definition of τ, $s'_{k'} = s$, for $\overline{\mathtt{s\_j}}$ representing state s with $\sigma(\overline{\mathtt{s\_j}}_{k'}) = \mathtt{tt}$. That means $s'_{k'}$ is the state being represented by the unique variable with value $\mathtt{tt}$. Thus,

$$s_{k'} = s'_{k'}$$

- For the absolute time reference z:

$$\nu_{k'}(z) = \sigma_t(\overline{\mathtt{z}}_{k'}) = \overset{def.\varsigma}{=} \nu'_{k'}(z)$$

- For all clocks $x \in X$:

$$\nu_{k'}(x) = \sigma_t(\overline{\mathtt{z}}_{k'}) - \big(\sigma_t(\overline{\mathtt{z}}_{k'}) - \nu_{k'}(x)\big) \overset{def.\varsigma}{=} \sigma_t(\overline{\mathtt{z}}_{k'}) - \sigma_t(\overline{\mathtt{x}}_{k'}) \overset{def.\tau}{=} \nu'_{k'}(x)$$

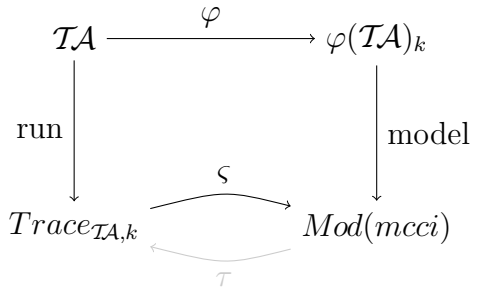


Figure 3.8.: Correctness of Representation

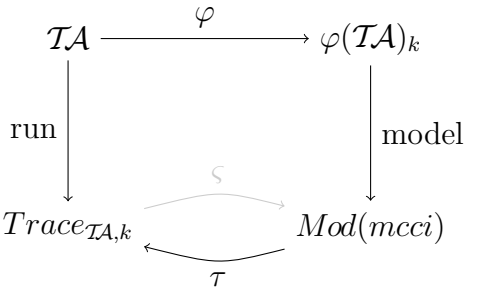


Figure 3.9.: Completeness of Representation

Thus,

$$\tau \circ \varsigma = id.$$

□

Proposition 3.3.24 (Correctness of Representation)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1. For $k \in \mathbb{N}$, let $\varphi(\mathcal{A})_k$ be its k-unrolling, let $\Sigma(\varphi(\mathcal{A})_k)$ be the corresponding signature (see figure 3.8).

1. For every trace r_k of length k of automaton $\mathcal{A}$,

   $$r_k : (s_0, \nu_0) \xrightarrow{\alpha_0} (s_1, \nu_1) \xrightarrow{\alpha_1} (s_2, \nu_2) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_{k-1}} (s_k, \nu_k),$$

   the corresponding interpretation $\sigma_{r_k} \in Mod(mcci)$ is a model for the k-unrolling, that means

   $$\sigma_{r_k} \models \varphi(\mathcal{A})_k$$

2. For every interpretation $\sigma \in Mod(mcci)$ being model for the k-unrolling, there exists a trace r_k of length k of $\mathcal{A}$ such that $\sigma = \sigma_{r_k}$.

Proof.

1. The proof is done inductively on the unrolling depth k of $\varphi(\mathcal{A})_k$.

   IA: (Induction anchor, $k = 0$) By definition of corresponding interpretation (see definition 3.3.18 on page 52):

$\sigma_{r_k} \models \overline{\mathtt{s\_0}}_0$	for $\overline{\mathtt{s\_0}}$ representing the initial state s_0
$\sigma_{r_k} \models \neg\overline{\mathtt{s\_j}}_0$	for $\overline{\mathtt{s\_j}}$ representing state $s_j \neq s_0$
$\sigma_{r_k} \models (\overline{\mathtt{x}}_0 = 0)$	for $\overline{\mathtt{x}}$ representing clock $x \in X$
$\sigma_{r_k} \models (\overline{\mathtt{z}}_0 = 0)$	for $\overline{\mathtt{z}}$ representing the absolute time reference z

   From the above follows

   $$\sigma_{r_k} \models \varphi_i(\mathcal{A}).$$

By definition of corresponding interpretation, also $\sigma_{r_k} \models \varphi_{gc}(\mathcal{A})_0$ (see remark 3.3.19), and thus σ_{r_k} is a model for the 0-unrolling of automaton $\mathcal{A}$,

$$\sigma_{r_k} \models \varphi(\mathcal{A})_0$$

IH: (Induction hypothesis) σ_{r_k} is a model for the k-unrolling $\varphi(\mathcal{A})_k$ of automaton $\mathcal{A}$, $\sigma_{r_k} \models \varphi(\mathcal{A})_k$.

IS: (Induction step from k to $k+1$)

– In case $\alpha_k = t \in \mathtt{Time}, t > 0$ (that means $(s_k, \nu_k) \xrightarrow{t} (s_{k+1}, \nu_{k+1})$ is a delay step), $s_{k+1} = s_k$ and $\nu_{k+1} = \nu_k + t$ by definition of automaton trace (see definition 2.4.11 on page 23). Then, by definition of corresponding interpretation,

$$\begin{aligned}
&\sigma_{r_k} \models \overline{\mathtt{s_k}}_{k+1}, && \text{for } \overline{\mathtt{s_k}} \text{ representing state } s_k \\
&\sigma_{r_k} \models (\overline{\mathtt{z}}_{k+1} = \overline{\mathtt{z}}_k + t) \\
&\sigma_{r_k} \models \bigwedge_{x \in X} (\overline{\mathtt{x}}_{k+1} = \overline{\mathtt{x}}_k) && \text{because} \\
& && \sigma(\overline{\mathtt{x}}_{k+1}) = \sigma(\overline{\mathtt{z}}_{k+1}) - \nu_{k+1}(x) \\
& && = \sigma(\overline{\mathtt{z}}_k) + t - (\nu_k(x) + t) \\
& && = \sigma(\overline{\mathtt{z}}_k) - \nu_k(x) = \sigma(\overline{\mathtt{x}}_k) \\
&\sigma_{r_k} \models I_k(\overline{\mathtt{s_k}}_k), && \text{(see remark 3.3.20)} \\
&\sigma_{r_k} \models I_{k+1}(\overline{\mathtt{s_k}}_{k+1}) && \text{(see remark 3.3.20)}
\end{aligned}$$

From the above follows $\sigma_{r_k} \models e_d(s_k)_{k+1}$ (see also (3.16)), thus by definition also $\sigma_{r_k} \models \varphi_E(\mathcal{A})_{k+1}$ (see also (3.18)). As $\sigma_{r_k} \models \varphi_{gc}(\mathcal{A})_{k+1}$ (see remark 3.3.19) and $\sigma \models \varphi_i(\mathcal{A})$ (see (1)), finally

$$\sigma_{r_k} \models \varphi(\mathcal{A})_{k+1}$$

– In case $\alpha_k = \tau$ (that means $(s_k, \nu_k) \xrightarrow{\tau} (s_{k+1}, \nu_{k+1})$ is an action transition), there exists a transition $(s_k, \varphi, Y, s_{k+1}) \in E$ such that $\nu_{k+1} = \nu_k[Y := 0]$. By definition of corresponding interpretation,

$$\begin{aligned}
&\sigma_{r_k} \models \overline{\mathtt{s_l}}_{k+1}, && \text{for } \overline{\mathtt{s_l}}_{k+1} \text{ representing state } s_{k+1} \\
&\sigma_{r_k} \models (\overline{\mathtt{z}}_{k+1} = \overline{\mathtt{z}}_k) \\
&\sigma_{r_k} \models \bigwedge_{x \notin Y} (\overline{\mathtt{x}}_{k+1} = \overline{\mathtt{x}}_k) && \text{for } \overline{\mathtt{x}} \text{ representing clock } x \text{ and} \\
& && \nu_{k+1}(x) = \nu_k(x) \\
&\sigma_{r_k} \models \bigwedge_{x \in Y} (\overline{\mathtt{x}}_{k+1} = \overline{\mathtt{z}}_{k+1}) && \text{for } \overline{\mathtt{x}} \text{ representing clock } x \text{ and } \nu_{k+1}(x) = 0 \\
&\sigma_{r_k} \models \varphi_k,
\end{aligned}$$

From the above follows $\sigma_{r_k} \models e_a((s_k, \varphi, Y, s_{k+1}))_{k+1}$ (see also (3.17)), thus by definition also $\sigma_{r_k} \models \varphi_E(\mathcal{A})_{k+1}$. As $\sigma_{r_k} \models \varphi_{gc}(\mathcal{A})_{k+1}$ and $\sigma \models \varphi_i(\mathcal{A})$, finally

$$\sigma_{r_k} \models \varphi(\mathcal{A})_{k+1}$$

2. Follows directly from lemma 3.3.23 for r_k is t_σ.

□

Proposition 3.3.25 (Completeness of Representation)
Let $\mathcal{A}$ be a timed automaton, using the notation of remark 3.3.1. For $k \in \mathbb{N}$, let $\varphi(\mathcal{A})_k$ be its k-unrolling, let $\Sigma(\varphi(\mathcal{A})_k)$ be the corresponding signature, let $\sigma \in Mod(mcci)$ be an interpretation, let t_σ be the corresponding trace of σ (see figure 3.9).

The corresponding trace t_σ is a trace of automaton $\mathcal{A}$, that means

$$t_\sigma \in Trace_{\mathcal{A},k}$$

Proof. This follows directly from lemma 3.3.23 and proposition 3.3.24. □

3.4. Formula Representation of Properties

As mentioned in 1.1 and 2.6, only safety properties like "A certain state s of the automaton can never be reached" will be considered here. This linguistic statement can be translated into the CTL formula $\neg\mathbf{EF}s$. As mentioned in 2.6, model checking will be done for the negation of the property, that means with $\mathbf{EF}s$.

Bounded model checking is done by unrolling the automaton (see section 2.6) up to a given depth, so the property also is checked up to this depth. That means, the property $\mathbf{EF}s$ to be verified no longer expressed as "s is eventually reachable", but instead as "s is eventually reachable within k steps". For $\overline{\mathbf{s}}$ being the representation of state s, the property then translates into a formula like

$$\overline{\mathbf{s}}_0 \vee \overline{\mathbf{s}}_1 \vee \overline{\mathbf{s}}_2 \vee \ldots \vee \overline{\mathbf{s}}_k$$

Note that it is not necessary to restrict the values of the other states, as this is done by the transition relation and the mutual exclusion constraint.

However, this translation of the property is true only for the first (complete) abstraction refinement loop. As soon as the property has been shown to be satisfied for unrolling depth k, it is also shown to be satisfied for all unrolling depths $k'' < k$, but nothing is known about the property for unrolling depths $k' > k$. Nevertheless, the result for unrolling depth k may be reused when checking unrolling depth $k' > k$: As the property is known to be satisfied for k and all unrolling depths smaller than k, it has to be unrolled only for those depths in between k' and k, that means

$$\overline{\mathbf{s}}_{k+1} \vee \overline{\mathbf{s}}_{k+2} \vee \ldots \vee \overline{\mathbf{s}}_{k'}$$

3.5. Formula Representation of a System of Automata

To obtain the transition relation for a system of timed automata, every automaton first is translated on its own. Thereby, the events have to be taken into account: To preserve the synchronisation strategy explained in sections 2.4.3 and 2.6.1, at most two transitions may fire at the same time in case they are labelled with input and output action of the same underlying channel. Thus, for every event a, two Boolean variables $a!$ and $a?$ are introduced.

According to the transition relation definition obtained from the single timed automata, $a!$ and $a?$ are added as conjunctive elements to the formula representation of an action transition (definition 3.3.5). Furthermore, the constraint

$$\bigwedge_{a \in \underline{A}} a! \leftrightarrow a?$$

is added to the formula representation $\varphi(\mathcal{A})$ of the automaton, to ensure either both actions representing one event hold at the same time or both do not hold.

To ensure appropriate firing of transitions within a system of timed automata, either one automaton does a transition labelled with the internal action τ while all other automata do delay steps, or two automata fire a transition each (labelled with input and output action of the same underlying channel) while all other automata do delay steps. To ensure this behaviour, several mutual exclusion constraints are needed:

Mutual exclusion of events: Every action transition must be labelled with an event from $A_{?!}$. For $A = \{e_1, \ldots, e_n\}$, it is enough to claim mutual exclusion for the output events:

$$\bigvee_{1 \le i \le n} (e_i! \wedge \bigwedge_{\substack{1 \le j \le n \\ j \neq i}} (\neg e_j!)),$$

as $e! \leftrightarrow e?$ assures it for the input events, too. This formula is needed once, containing all events appearing in the automata system, and is quadratic in the number of events. In this way, mutual exclusion of events is ensured, but for $a! \leftrightarrow a? \leftrightarrow \texttt{tt}$ it is not ensured that only one transition labelled with $a!$ and only one transition labelled with $a?$ fire at the same time. To ensure this, mutual exclusion of automata is needed in the following way:

Mutual exclusion of automata: One Boolean A variable per automaton $\mathcal{A}$ is introduced, with intended meaning that A holds iff automaton $\mathcal{A}$ does an action transition and all other do delay transitions. Furthermore, one Boolean variable AB for every ordered pair of automata ($\mathcal{A}$,$\mathcal{B}$) is introduced, with the intended meaning that AB holds iff automaton $\mathcal{A}$ does an output action transition, automaton $\mathcal{B}$ does an input action transition and all other automata do delay transitions. For n being the number of automata in the system, the number of variables thus is

$$\underbrace{n}_{\text{For single automaton}} + \underbrace{2 * \binom{n}{2}}_{\text{For pairs of automata}} = n + 2 * \frac{n * (n-1)}{2} = n^2$$

Only one of these variables must be true at every time, which leads to the following mutual exclusion formula. Let $\{A_1, \ldots, A_n\}$ be the set of variables for single automaton, let $\{A_{1'}, A_{2'}, \ldots, A_m\}$ (with $m = n * (n-1)$) be the set of variables for pairs of automata. The mutual exclusion formula is

$$\bigvee_{1 \leq i \leq n} (A_i \wedge \bigwedge_{\substack{1 \leq j \leq n \\ j \neq i}} (\neg A_j) \wedge \bigwedge_{1 \leq k \leq m} (\neg A_k)) \vee$$
$$\bigvee_{1 \leq i \leq m} (A_i \wedge \bigwedge_{\substack{1 \leq j \leq m \\ j \neq i}} (\neg A_j) \wedge \bigwedge_{1 \leq k \leq n} (\neg A_k)).$$

This formula is quadratic in the number of variables, thus in $O(n^4)$.

The transition relation of every automaton $\mathcal{A}$ is "divided" in four parts, and each part conjuncted with the appropriate automaton variables:

- Part one contains all action transition formulae labelled with an output action $a! \neq \tau$. This part is conjuncted with the disjunction of all variables XY where $X = A$.
- Part two contains all action transition formulae labelled with an input action $a? \neq \tau$. This part is conjuncted with the disjunction of all variables XY where $Y = A$.
- Part three contains all action transition formulae labelled with the internal action τ. This part is conjuncted with A.
- Part four contains all delay transition formulae. This part is conjuncted with $\neg A \wedge \neg AY \wedge \neg XA$ for all variables AY and XA containing A.

In this way, mutual exclusion of transitions is ensured, and the synchronisation strategy explained in sections 2.4.3 and 2.6.1 is preserved.

Example 3.5.1 (Mutual Exclusion of Automata)
Consider the automata as depicted in figure 3.10.

Let s_1 and s_2 be states in automaton A, t_1 and t_2 be states in automaton B and u_1 and u_2 be states in automaton C. These three automata are translated with the formula depicted in figure 3.11. Although the formula is quite long, the large size of the example is chosen for explanatory purposes.

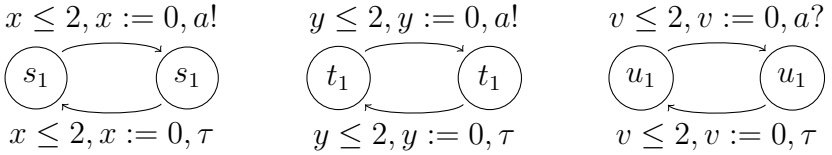


Figure 3.10.: Example of mutual exclusion of automata

□

$$
\begin{gathered}
\big[(A \wedge \neg B \wedge \neg C \wedge \neg AB \wedge \neg AC \wedge \neg BC \wedge \neg BA \wedge \neg CA \wedge \neg CB) \vee \\
(\neg A \wedge B \wedge \neg C \wedge \neg AB \wedge \neg AC \wedge \neg BC \wedge \neg BA \wedge \neg CA \wedge \neg CB) \vee \\
(\neg A \wedge \neg B \wedge C \wedge \neg AB \wedge \neg AC \wedge \neg BC \wedge \neg BA \wedge \neg CA \wedge \neg CB) \vee \\
(\neg A \wedge \neg B \wedge \neg C \wedge AB \wedge \neg AC \wedge \neg BC \wedge \neg BA \wedge \neg CA \wedge \neg CB) \vee \\
(\neg A \wedge \neg B \wedge \neg C \wedge \neg AB \wedge AC \wedge \neg BC \wedge \neg BA \wedge \neg CA \wedge \neg CB) \vee \\
(\neg A \wedge \neg B \wedge \neg C \wedge \neg AB \wedge \neg AC \wedge BC \wedge \neg BA \wedge \neg CA \wedge \neg CB) \vee \\
(\neg A \wedge \neg B \wedge \neg C \wedge \neg AB \wedge \neg AC \wedge \neg BC \wedge BA \wedge \neg CA \wedge \neg CB) \vee \\
(\neg A \wedge \neg B \wedge \neg C \wedge \neg AB \wedge \neg AC \wedge \neg BC \wedge \neg BA \wedge CA \wedge \neg CB) \vee \\
(\neg A \wedge \neg B \wedge \neg C \wedge \neg AB \wedge \neg AC \wedge \neg BC \wedge \neg BA \wedge \neg CA \wedge CB)\big] \\
\wedge \\
(a! \leftrightarrow a?) \\
\wedge \\
(a! \wedge \neg\tau) \vee (\neg a! \wedge \tau) \\
\wedge \\
\big[((s_{1_t} \wedge \neg s_{2_t}) \vee (\neg s_{1_2} \wedge s_{2_t})) \wedge \\
((t_{1_t} \wedge \neg t_{2_t}) \vee (\neg t_{1_t} \wedge t_{2_t})) \wedge \\
((u_{1_t} \wedge \neg u_{2_t}) \vee (\neg u_{1_t} \wedge u_{2_t}))\big] \\
\wedge \\
\big[(s_{1_t} \wedge (z_t - x_t \leq 2) \wedge a!_t \wedge s_{2_{t+1}} \wedge (z_{t+1} - x_{t+1} = 0) \wedge (AB_t \vee AC_t)) \vee \\
(s_{1_t} \wedge (z_t - x_t \leq 2) \wedge \tau_t \wedge s_{2_{t+1}} \wedge (z_{t+1} - x_{t+1} = 0) \wedge A) \vee \\
(s_{1_t} \wedge s_{1_{t+1}} \wedge (x_{t+1} = x_t) \wedge (\neg A \wedge \neg AB \wedge \neg AC \wedge \neg BA \wedge \neg CA)) \vee \\
(t_{1_t} \wedge (z_t - y_t \leq 2) \wedge a!_t \wedge t_{2_{t+1}} \wedge (z_{t+1} - y_{t+1} = 0) \wedge (BA_t \vee BC_t)) \vee \\
(t_{1_t} \wedge (z_t - y_t \leq 2) \wedge \tau_t \wedge t_{2_{t+1}} \wedge (z_{t+1} - y_{t+1} = 0) \wedge B) \vee \\
(t_{1_t} \wedge t_{1_{t+1}} \wedge (y_{t+1} = y_t) \wedge (\neg B \wedge \neg AB \wedge \neg BC \wedge \neg BA \wedge \neg CB)) \vee \\
(u_{1_t} \wedge (z_t - v_t \leq 2) \wedge a?_t \wedge u_{2_{t+1}} \wedge (z_{t+1} - v_{t+1} = 0) \wedge (AC_t \vee BC_t)) \vee \\
(u_{1_t} \wedge (z_t - v_t \leq 2) \wedge \tau_t \wedge u_{2_{t+1}} \wedge (z_{t+1} - v_{t+1} = 0) \wedge C) \vee \\
(u_{1_t} \wedge u_{1_{t+1}} \wedge (v_{t+1} = v_t) \wedge (\neg C \wedge \neg AC \wedge \neg BC \wedge \neg CA \wedge \neg CB))\big]
\end{gathered}
$$

Figure 3.11.: Formula obtained for the automata in figure 3.10

4. Abstraction

4.1. Overview

Real time systems used for example in industry to control complex operations themselves become very complex. Therefor, it is of great importance to ensure correct functionality of the systems even before they are installed. To ensure this, verification techniques and model checking are used, but due to the increasing size of real time systems, the amount of time required to return correct results also increases (remember the state explosion problem explained in section 1.1). Abstraction techniques help in dealing with this problem.

In general, abstraction means to remove information from a system which is not relevant to the property to be satisfied. The system is viewed as a large collection of constraints, and abstraction as removing irrelevant constraints, with the goal of eliminating variables that occur only in the irrelevant constraints. After having found the irrelevant constraints, verification and model checking are performed for the reduced system, without—or at least with marginal—information loss.

One of the main problems for abstraction thus is how to defined the abstraction function. An abstraction function defined to fine (that means only few parameters are abstracted) will lead to state explosion, while an abstraction function defined too coarse will lead to great information loss (in fact, abstraction always means information loss, but with regard to the property, there exist facts which are not as relevant as others). Abstraction techniques therefor may be distinguished by how they deal with this information loss:

- *Over-approximation techniques* (such as predicate abstraction) release constraints which enriches the behaviour of the system (in that more traces are possible). Correctness of the abstract system (that means the property is satisfied in the abstract system) implies correctness of the concrete system. Abstraction techniques providing this behaviour are also called *conservative techniques*. Due to allowing more traces, over-approximation techniques admit false negatives, that means spurious counterexamples: A false negative is a trace which violates the property in the abstract system (and thus is a counterexample), but does not violate the property in the concrete system (and thus is spurious). The abstract system is also said to *simulate* the concrete system.

- *Under-approximation techniques* remove irrelevant behaviour from the system which constrains the possible behaviour (in that fewer traces are possible). Falseness of the abstract system (that means the property is not satisfied in the abstract system) implies falseness of the concrete system. Due to allowing fewer traces, under-approximation techniques admit false positives: A false

positive is found in case the abstract systems claims the property to be satisfied, but there exists a trace in the concrete system violating the trace.

Both false positives and false negatives result from information loss in the abstract system, compared to the concrete system. In either case, as the abstraction returns incorrect results, it has to be refined, that means the abstraction from a (set of or single) parameter has to be undone. Over-approximation techniques will be used within the context of this thesis, as it is the aim to verify properties (and not refute them, this would be done by using under-approximation techniques). For a more detailed description of abstraction techniques, see for example (Clarke *et al.*, 1999) or (Clarke *et al.*, 2003).

The question arises whether an abstraction function is correct with regard to over- or under-approximation or not. As the name suggests, to be a correct over-approximation function, the abstraction has to ensure the semantics of the abstract system is a (not necessarily proper) superset of the semantics of the concrete system. That means, on the other hand, a behaviour which is not contained in the semantics of the abstract system must not be contained in the semantics of the concrete system[1]. Thus, to prove an (over-approximation) abstraction function to be correct, it has to be shown every possible behaviour of the concrete system is contained in the possible behaviour of the abstract system.

4.2. Abstraction of Timed Automata

The abstraction of timed automata going to be presented here works on the formula representation presented in chapter 3. Roughly speaking, the abstraction is done by modifying the formulae representation. The exact modification (formula transformation) thereby depends on the type of parameters intended to be abstracted (clocks, events, etc).

The finite set of parameters to abstract from will be called *abstraction set*, denoted by $\mathcal{AS}$. Parameters of timed automata in formula representation are given by means of variables, thus, the abstraction set will contain variables, that means $\mathcal{AS} \subset \mathcal{V} \cup \mathcal{P}$. During the abstraction refinement cycle, $\mathcal{AS}$ will be modified (by adding/removing parameters), depending on the model checking results of preceeding steps. To start the verification, the initial abstraction set has to be chosen manually, depending on the model to be checked, its supposed behaviour, the property to be verified and some external knowledge giving advise which parameters might be needed (for example heuristic observations regarding the usage of a certain clock).

Remember for any basic component of a timed automaton $\mathcal{A}$, there will be $k+1$ variables representing this component in the k-unrolling $\varphi(\mathcal{A})_k$ of $\mathcal{A}$ (one variable for every unrolling depth), while there are up to three variables representing this component in the formula representation $\varphi(\mathcal{A})$[2]. If for example x is a clock of

[1] The exact meaning of "semantics" and "behaviour" depends on the underlying system: For timed automata, the semantics is defined via trace sets, while for example for formulae, the semantics is defined via an interpretation.

[2] For any basic component of a timed automaton $\mathcal{A}$, the formula representation $\varphi(\mathcal{A})$ contains one variable subscripted with t and one variable subscripted with $t+1$. Furthermore, for clocks and

some automaton $\mathcal{A}$, the k-unrolling will contain the variables $\overline{\mathtt{x}}_0, \overline{\mathtt{x}}_1, \ldots, \overline{\mathtt{x}}_k$, while the formula representation will contain the variables $\overline{\mathtt{x}}_t, \overline{\mathtt{x}}_{t+1}$ and $\overline{\mathtt{x}}_0$. For ease of notation, the abstraction set shall only contain variable x (with the intended meaning to abstract from x by abstracting from all variables $\overline{\mathtt{x}}_0, \overline{\mathtt{x}}_1, \ldots, \overline{\mathtt{x}}_k$ in $\varphi(\mathcal{A})_k$ and from the variables $\overline{\mathtt{x}}_t, \overline{\mathtt{x}}_{t+1}, \overline{\mathtt{x}}_0$ in $\varphi(\mathcal{A})$).

4.2.1. Abstract from Clocks

Properties considered within the context of this thesis are of the form $q = \mathbf{EF}s$, expressing the question whether state s is reachable or not. The most simple solution to this question surely is given in case s is not even reachable via the transition relation, that means s is for example an isolated state. As for these cases, reachability analysis is quite easy and may be done by means of graph algorithms (and hence especially without need for formal model checking), they shall not be considered in the sequel.

If reachability analysis is done for a single automaton, clock abstraction is enough to gain correct results, as by removing all clocks, everything restricting the reachability of s is removed (remember that for a single automaton, all transitions are labelled with the internal action τ, as there is no possibility to synchronise). The strategy going to be used for clock abstraction while taking this into account will be called *abstraction by omission*.

In general, abstraction by omission simply deletes (while preserving a well-defined formula structure according to definition 2.1.2) every occurrence of the parameters from the abstraction set $\mathcal{AS}$. In more detail, this will (conceptually) be done by substituting every atomic formula containing a parameter from $\mathcal{AS}$ (clocks in this case) with `true`.

For the purpose of clarity, the abstraction will first be presented twofold: The first version is very simple and does not provide any simplifications (which are surely possible for formulae containing `true` as subformula). The proof of correctness benefits form this version with regard to simplicity and clarity. Afterwards, an improved version of the abstraction by omission is presented, together with a proof that both versions result in semantically equivalent formulae.

Abstraction by omission is not reasonable for arbitrary parameters, for more details on these restrictions, please refer to page 82.

Definition 4.2.1 (Abstraction by Omission, Simple Form)
Let $\mathcal{A}$ be a timed automaton, let $\varphi(\mathcal{A})$ and $\varphi(\mathcal{A})_k$ be its formula representation and k-unrolling for some $k \in \mathbb{N}$, respectively. Let both $\varphi(\mathcal{A})$ and $\varphi(\mathcal{A})_k$ be in negation normal form (see remark 3.3.14), let $\mathcal{AS}$ be the abstraction set.

The *abstraction of* $\varphi(\mathcal{A})$ *from* $\mathcal{AS}$ *by omission*, denoted by $\alpha(\varphi(\mathcal{A}))$, is defined by applying transformation function α (depicted in figure 4.1) to $\varphi(\mathcal{A})$. The *abstraction of* $\varphi(\mathcal{A})_k$ *from* $\mathcal{AS}$ *by omission*, denoted by $\alpha(\varphi(\mathcal{A})_k)$, is defined in the same way.

To avoid ambiguities regarding the abstraction set, $\alpha(\varphi(\mathcal{A}))$ and $\alpha(\varphi(\mathcal{A})_k)$ may also be written as $\alpha(\varphi(\mathcal{A}))_{-\mathcal{AS}}$ and $\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}$, respectively. □

the initial state, $\varphi(\mathcal{A})$ contains a third variable representing the component which is subscripted with 0 (see also section 3.3)

$$\alpha : \mathcal{F}(\mathcal{P}, \mathcal{V})_{nnf} \rightarrow \mathcal{F}(\mathcal{P} \setminus \mathcal{AS}, \mathcal{V} \setminus \mathcal{AS})_{nnf}$$

$$\alpha(L) = \begin{cases} L & \Leftarrow \Sigma(L) \cap \mathcal{AS} = \emptyset \\ \texttt{true} & \Leftarrow \Sigma(L) \cap \mathcal{AS} \neq \emptyset \end{cases}$$
$$\alpha(F \wedge G) = \alpha(F) \wedge \alpha(G)$$
$$\alpha(F \vee G) = \alpha(F) \vee \alpha(G)$$

Let F and G be formulae in negation normal form, let L be a literal (see page 8), let $\Sigma(L)$ be the corresponding signature (see definition 2.1.11), let $\mathcal{AS}$ be the abstraction set. Note that $\alpha(F) = F$ in case $\mathcal{AS} \cap \Sigma(F) = \emptyset$, that means if a formula F does not contain variables appearing in the abstraction set, the abstraction function α returns the identity.

Figure 4.1.: Abstraction by Omission, Simple Form

Correctness Proof

Abstraction consists in omitting constraints and thus "enriching" the semantics of the system. An abstraction thus is considered to be correct if the abstract system does not restrict the semantics/behaviour of the concrete system. Roughly speaking, the possible behaviour of the abstract system is a (proper) superset of the possible behaviour of the concrete system.

What "superset" means depends on the system which is abstracted. In this case, where the sematics of timed automata is defined via trace sets, superset means the set of possible traces for the abstract automaton is a superset of the set of possible traces of the concrete automaton.

The intuitive idea why abstraction by omission results in a correct abstraction is: By omitting clock constraints, invariants and transition guards will hold more often, so the possible behaviour is enriched.

To formally ensure the semantics is not reduced when defining an abstraction function, a correspondance between concrete and abstract system has to be found, expressing the concept of preserving the semantics and behaviour. This correspondance will be defined with homomorphisms, mapping from the concrete to the abstract system.

As the behaviour is characterised by set of possible traces (definition 2.4.11 on page 23), a homomorphism first is defined between sets of traces. Afterwards, a stronger correspondance may be defined by a homomorphism between transition systems and even last, a homomorphism between timed automata is defined expressing syntactical correspondance.

Definition 4.2.2 (Homomorphism of Traces)
Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two timed automata with $\underline{A}_1 = \underline{A}_2$ and $X_2 \subseteq X_1$. Let $S_{\mathcal{A}_1}$ and $S_{\mathcal{A}_2}$ be the associated transition systems, let $Trace_{\mathcal{A}_1}$ and $Trace_{\mathcal{A}_2}$ be the sets of

traces of $\mathcal{A}_1$ and $\mathcal{A}_2$.

A function h_T:$Trace_{\mathcal{A}_1} \rightarrow Trace_{\mathcal{A}_2}$ is called a *homomorphism of traces (between $Trace_{\mathcal{A}_1}$ and $Trace_{\mathcal{A}_2}$)* iff for each trace

$$t_1 : (s_0, \nu_0) \stackrel{\alpha_0}{\rightarrow} (s_1, \nu_1) \stackrel{\alpha_1}{\rightarrow} (s_2, \nu_2) \ldots \in Trace_{\mathcal{A}_1},$$

there exists a trace $h(t_1) = t_2$,

$$t_2 : (\widetilde{s_0}, \nu_0') \stackrel{\alpha_0}{\rightarrow} (\widetilde{s_1}, \nu_1') \stackrel{\alpha_1}{\rightarrow} (\widetilde{s_2}, \nu_2') \ldots \in Trace_{\mathcal{A}_2},$$

with $\nu_i' = \nu_i|_{X_2}$ (ν_i' is the restriction of ν_i to X_2, as defined in definition 2.4.5 on page 20) and $\alpha_i \in \mathtt{Time} \cup \underline{A}_{1,?!}$, for all $i \geq 0$.

For the sets of finite traces $Trace_{\mathcal{A}_1,k}$ and $Trace_{\mathcal{A}_2,k}$, h_T is defined analogously. □

According to this definition, for each trace in $Trace_{\mathcal{A}_1}$, there exists a trace in $Trace_{\mathcal{A}_2}$ with the same observable behaviour (see definition 2.4.11), that means the same sequence of labels. Note that indirections[3] are not allowed at first glance, in that for every trace in $Trace_{\mathcal{A}_1}$, h_T requires (the existence of) a trace in $Trace_{\mathcal{A}_2}$ with exactly the same observable behaviour. The homomorphism of traces thus is equivalent to the simulation idea, in that it claims trace inclusion (that means inclusion of trace sets).

Nevertheless, indirections are possible: Consider a trace t_1 in $Trace_{\mathcal{A}_1}$ and a trace t_2, such that t_2 is an indirection with regard to t_1. t_2 may be contained in $Trace_{\mathcal{A}_2}$ even though there is no trace t_2' in $Trace_{\mathcal{A}_1}$ which is mapped to t_2 by h_T. This is due to the fact that h_T is not necessarily surjective, as the definition of homomorphism does not claim equality but inclusion of trace sets.

The homomorphism of traces h_T defines a correspondence between trace sets of timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$. It is also possible to define a homomorphism between the associated transition systems, which is a stronger correspondence that the homomorphism of traces.

Definition 4.2.3 (Homomorphism of Transition Systems)
Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two timed automata with $\underline{A}_1 = \underline{A}_2$, let $X_2 \subseteq X_1$. Let $S_{\mathcal{A}_1} = (Q_{\mathcal{A}_1}, (s_0, \nu_0), \mathtt{Time} \cup \underline{A}_{1,?!}, \rightarrow_1)$ and $S_{\mathcal{A}_2} = (Q_{\mathcal{A}_2}, (\widetilde{s_0}, \widetilde{\nu_0}), \mathtt{Time} \cup \underline{A}_{2,?!}, \rightarrow_2)$ be the associated transition systems.

A function h_L:$S_{\mathcal{A}_1} \rightarrow S_{\mathcal{A}_2}$—which consists of two mappings h_L:$Q_{\mathcal{A}_1} \rightarrow Q_{\mathcal{A}_2}$ and h_L:$(\rightarrow_1) \rightarrow (\rightarrow_2)$—is called a *homomorphism of transition systems*[4] *(between $S_{\mathcal{A}_1}$ and $S_{\mathcal{A}_2}$)* iff

1. $h_L((s_0, \nu_0)) = (\widetilde{s_0}, \widetilde{\nu_0})$, for $\widetilde{\nu_0} = \nu_0|_{X_2}$,
2. for each configuration (s, ν) with $h_L((s, \nu)) = (\widetilde{s}, \widetilde{\nu})$: $\widetilde{\nu} = \nu|_{X_2}$, that means the valuations agree on common clock variables,

[3] Consider a trace t_1 with $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \ldots)$ being the sequence of its labels, and a trace t_2 with $(\alpha_1', \alpha_2', \alpha_3', \alpha_4', \ldots)$ being the sequence of its labels. t_1 is an indirection with regard to t_2 if $(\alpha_1', \alpha_2', \alpha_3', \alpha_4', \ldots)$ is a subsequence of $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \ldots)$.

[4] Formally, there is a mapping h_Q:$Q_{\mathcal{A}_1} \rightarrow Q_{\mathcal{A}_2}$ and a mapping $h_{\rightarrow}$:$(\rightarrow_1) \rightarrow (\rightarrow_2)$ such that $h_L = h_Q \oplus h_{\rightarrow}$, but, for convenience, the above abuse of notation shall be used.

3. for each delay transition $(s, \nu) \xrightarrow{t} (s', \nu')$ in $\rightarrow_1$, there exists a delay transition $h_L((s, \nu)) \xrightarrow{t} h_L((s', \nu'))$ in $\rightarrow_2$ and

4. for each action transition $(s, \nu) \xrightarrow{\alpha} (s', \nu')$ in $\rightarrow_1$, there exists an action transition $h_L((s, \nu)) \xrightarrow{\alpha} h_L((s', \nu'))$ in $\rightarrow_2$.

□

In contrast to h_T, h_L not only claims the same observable behaviour but also a structural correspondence between the associated transition systems. Indirections are possible with the same argumentation as above.

Definition 4.2.4 (Homomorphism of Timed Automata)
Let $\mathcal{A}_1 = (\underline{A}, S_1, s_0, X_1, I_1, E_1)$ and $\mathcal{A}_2 = (\underline{A}, S_2, \widetilde{s_0}, X_2, I_2, E_2)$ be two timed automata with $X_2 \subseteq X_1$.

A function h_A:$\mathcal{A}_1 \rightarrow \mathcal{A}_2$, which consists of mappings h_A:$S_1 \rightarrow S_2$ and h_A:$E_1 \rightarrow E_2$ is called a *homomorphism of timed automata (between $\mathcal{A}_1$ and $\mathcal{A}_2$)* iff

1. $h_A(s_0) = \widetilde{s_0}$.

2. $\models I_1(s) \rightarrow I_2(h_A(s))$ for all states $s \in S_1$

3. For all states $s, s' \in S_1$ and all transitions $(s, \alpha, \varphi, Y, s') \in E_1$, there exists a transition $(h_A(s), \alpha, \widetilde{\varphi}, \widetilde{Y}, h_A(s')) \in E_2$ such that $\models \varphi \rightarrow \widetilde{\varphi}$ and $\widetilde{Y} \subseteq Y$.

□

Proposition 4.2.5 (Relationship of Homomorphisms)
The homomorphisms defined in definition 4.2.2, 4.2.3 and 4.2.4 are consecutive specialisations:

1. Every homomorphism h_A of timed automata according to definition 4.2.4 induces a unique homomorphism of transition systems according to definition 4.2.3.

2. Every homomorphism h_L of transition systems according to definition 4.2.3 induces a unique homomorphism of traces according to definition 4.2.2.

Proof.

1. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two timed automata with $\underline{A}_1 = \underline{A}_2$ and $X_2 \subseteq X_1$. Let $S_{\mathcal{A}_1} = (Q_{\mathcal{A}_1}, (s_0, \nu_0), \texttt{Time} \cup \underline{A}_{1,?!}, \rightarrow_1)$ and $S_{\mathcal{A}_2} = (Q_{\mathcal{A}_2}, (\widetilde{s_0}, \widetilde{\nu_0}), \texttt{Time} \cup \underline{A}_{2,?!}, \rightarrow_2)$ be the associated transition systems.

   The homomorphism h_L of transition systems, $h_L : S_{\mathcal{A}_1} \rightarrow S_{\mathcal{A}_2}$ (see also definition 4.2.3), is defined as

   1. $h_L((s_0, \nu_0)) = (h_A(s_0), \widetilde{\nu_0})$, with $\widetilde{\nu_0} = \nu|_{X_2}$. $(h_A(s_0), \widetilde{\nu_0})$ is a valid configuration (that means $\widetilde{\nu_0}$ satisfies the invariant of $h_A(s_0)$), as $h_A(s_0) = \widetilde{s_0}$, and $\widetilde{\nu_0}(x) = 0$ for all clocks $x \in X_2$ by definition.

2. $h_L((s,\nu)) = (h_A(s), \widetilde{\nu})$, with $\widetilde{\nu} = \nu|_{X_2}$. $(h_A(s), \widetilde{\nu})$ is a valid configuration, as $\models I_1(s) \rightarrow I_2(h_A(s))$ and $\nu(x) = \widetilde{\nu}(x)$ for all clocks $x \in X_2$.
3. For each delay transition $(s,\nu) \xrightarrow{t} (s',\nu')$ in $\rightarrow_1$, the delay transition $h_L((s,\nu)) \xrightarrow{t} h_L((s',\nu'))$ in $\rightarrow_2$ is defined as $(h_A(s), \nu|_{X_2}) \xrightarrow{t} (h_A(s'), \nu'|_{X_2})$: According to lemma 2.4.7 (convex invariants), this delay step is well-defined.
4. For each action transition $(s,\nu) \xrightarrow{\alpha} (s',\nu')$ in $\rightarrow_1$, the action transition $h_L((s,\nu)) \xrightarrow{\alpha} h_L((s',\nu'))$ in $\rightarrow_2$ is defined as $(h_A(s), \nu|_{X_2}) \xrightarrow{\alpha} (h_A(s'), \nu'|_{X_2})$: By definition of $S_{\mathcal{A}_1}$, there exists a transition $(s, \alpha, \varphi, Y, s') \in E_1$ such that $\nu \models \varphi$ and $\nu' = \nu[Y := 0]$. By definition 4.2.4, there exists a transition $(h_A(s), \alpha, \widetilde{\varphi}, \widetilde{Y}, h_A(s')) \in E_2$ such that $\nu|_{X_2} \models \widetilde{\varphi}$ and $\nu'|_{X_2} = \nu|_{X_2}[\widetilde{Y} := 0]$, and the action transition is well-defined.

2. Directly follows from definition 4.2.3.

□

Roughly speaking, an abstraction is correct if the semantics of the abstract system is not reduced with regard to the concrete system, and thus every possible behaviour of the concrete system also is a possible behaviour in the abstract system. As the semantics of timed automata is defined via trace sets (see definition 2.4.11), an abstraction of timed automata is correct if for all timed automata $\mathcal{A}$ and $\widetilde{\mathcal{A}}$, such that $\widetilde{\mathcal{A}}$ is obtained from $\mathcal{A}$ by abstraction, there exists a homomorphism of traces

$$h_T{:}Trace_{\mathcal{A},k} \rightarrow Trace_{\widetilde{\mathcal{A}},k},$$

as defined in definition 4.2.2. Thus, for the abstraction by omission to be correct, the abstraction function α has to be proved to preserve this property.

To formally prove the correctness, figure 4.2 is shown to be a commuting diagram. Therefor, recall the following definitions: $\mathcal{TA}$ denotes the set of all timed automata (definition 2.4.3), $\varphi(\mathcal{TA})_k$ denotes the set of all k-unrollings (definition 3.3.11), $Trace_{\mathcal{TA},k}$ denotes the set of all finite traces of length k (definition 2.4.11), $Mod(mcci)$ denotes the set of interpretations satisfying the constraints MCCI (equation (3.24)).

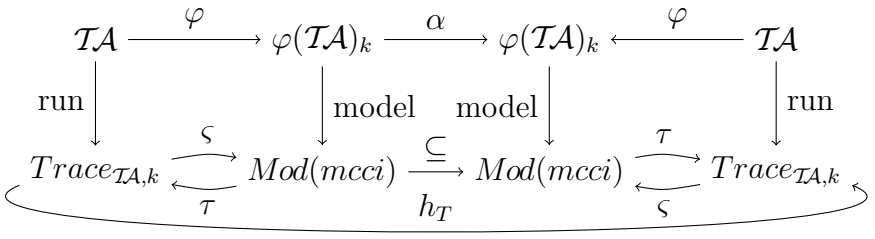


Figure 4.2.: Overview of Abstraction by Omission: Commuting diagram

Theorem 4.2.6 (Correctness of Abstraction by Omission)
Let $\mathcal{AS} \subseteq X$ be an abstraction set containing only clocks.

Using this, there exists a homomorphism of traces h_T such that figure 4.2 is a commuting diagram, that means the abstraction by omission is correct. □

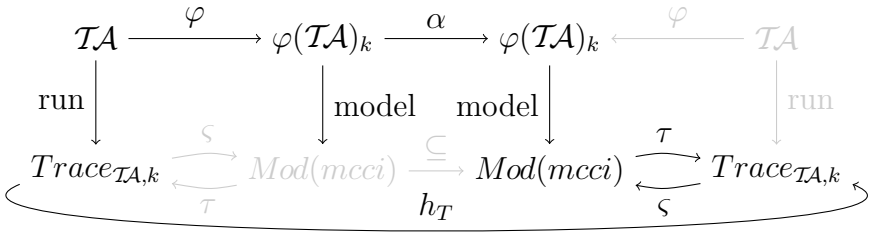


Figure 4.3.: Abstraction by Omission: Basic proof idea

The basic idea of the proof is the following (for explanatory purposses of this idea, please consider figure 4.3): For an automaton $\mathcal{A}$ and its k-unrolling $\varphi(\mathcal{A})_k$, α preserves the k-unrolling, that means the abstraction of $\varphi(\mathcal{A})_k$ from $\mathcal{AS} \subseteq X$ by omission, $\alpha(\varphi(\mathcal{A})_k)$, also is the k-unrolling of some automaton $\widetilde{\mathcal{A}}$ (upper part of figure 4.3)[5]. Although the abstraction function α does not return an automaton but a formula representation, this is of little importance, as automaton $\widetilde{\mathcal{A}}$ may be "derived" from the formula representation $\alpha(\varphi(\mathcal{A})_k)$. Proposition 4.2.10 shows the existence of $\widetilde{\mathcal{A}}$. For t being a trace in $\mathcal{A}$ and t' being a trace in $\widetilde{\mathcal{A}}$, such that $h_T(t) = t'$ (leftmost and lower part of figure 4.3), there exists an interpretation σ, being model for the k-unrolling of $\widetilde{\mathcal{A}}$, such that t' is the corresponding trace t_σ of σ (and thus, by lemma 3.3.23, also σ is the corresponding interpretation $\sigma_{t'}$ of t'). The existence of this homomorphism will be shown in the proof of theorem 4.2.6.

In other words: The possible behaviour of the abstract automaton $\widetilde{\mathcal{A}}$ (represented by the set of traces $Trace_{\widetilde{\mathcal{A}},k}$) is obtained from the possible behaviour (the set of traces $Trace_{\mathcal{A},k}$) of the concrete automaton $\mathcal{A}$ and the homomorphism of traces h_T. $Trace_{\widetilde{\mathcal{A}},k}$ also is obtained from the k-unrolling $\varphi(\mathcal{A})_k$ of $\mathcal{A}$, the abstraction function α, the set of interpretations being model for $\alpha(\varphi(\mathcal{A})_k)$ and the set of corresponding traces for these interpretations. Thus, the abstraction by omission is correct for clocks.

In addition to those parts depicted in figure 4.3, the other parts of figure 4.2 will be shown to form a commuting diagram, too.

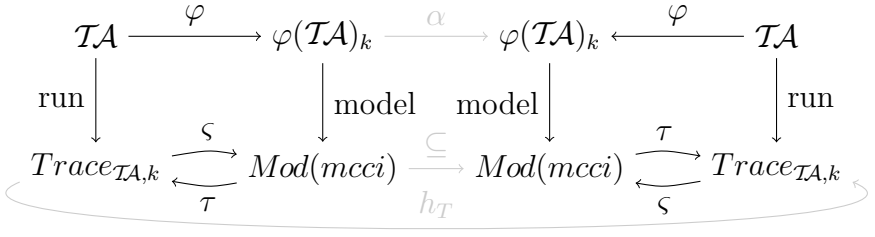


Figure 4.4.: Abstraction by Omission: First Commuting Subdiagram

Remark 4.2.7 (Commuting Subdiagram)
The two subparts of figure 4.2 depicted in figure 4.4 represent a commuting diagram each. This follows directly from propositions 3.3.24 and 3.3.25 in section 3.3.4. □

[5]In the same way, for $\varphi(\mathcal{A})$ being the formula representation of automaton $\mathcal{A}$, the abstraction of $\varphi(\mathcal{A})$ from $\mathcal{AS} \subseteq X$ by omission is the formula representation of some automaton $\widehat{\mathcal{A}}$.

The right part of figure 4.4 is a commuting diagram when considered seperately (according to propositions 3.3.24 and 3.3.25). Nevertheless, with regard to the overall context of figure 4.2, the fact that the abstraction function does not return an automaton but a formula (as mentioned above) has to be taken into account, as—strictly speaking—the phrase "commuting diagram" is not fully correct then. But due to the fact that proposition 4.2.10 will prove the existence of an appropriate timed automaton, using the expression "commuting diagram" is justifiable.

Lemma 4.2.8 (Weakening through Abstraction by Omission)
Let F be a formula in negation normal form, let $\alpha(F)$ be the abstract formula obtained from abstraction by omission, with abstraction set $\mathcal{AS} \subseteq \Sigma(F)$.

The abstraction by omission $\alpha(F)$ is a conservative approximation of F, that means $\alpha(F)$ is weaker that F and

$$\models F \rightarrow \alpha(F). \tag{4.1}$$

Proof. Let L be a literal. The proof is done inductively on the structure of F:

IA: $F = L$:

- If $\Sigma(L) \cap \mathcal{AS} = \emptyset$, then $\alpha(F) = L$. ($\models L \rightarrow L$) holds trivially.
- If $\Sigma(L) \cap \mathcal{AS} \neq \emptyset$, then $\alpha(F) = \texttt{true}$. ($\models L \rightarrow \texttt{true}$) holds trivially.

IH: For formulae F_1 and F_2 being subformulae of F, let ($\models F_1 \rightarrow \alpha(F_1)$) and ($\models F_2 \rightarrow \alpha(F_2)$).

IS: $F = F_1 \wedge F_2$:

$\alpha(F) = \alpha(F_1 \wedge F_2) = \alpha(F_1) \wedge \alpha(F_2)$ by definition of α. By IH and propositional logic (definition of $\wedge$ and $\rightarrow$):

$$\models (F_1 \wedge F_2) \rightarrow (\alpha(F_1) \wedge \alpha(F_2))$$

$F = F_1 \vee F_2$:

$\alpha(F) = \alpha(F_1 \vee F_2) = \alpha(F_1) \vee \alpha(F_2)$ by definition of α. By IH and propositional logic (definition of $\vee$ and $\rightarrow$):

$$\models (F_1 \vee F_2) \rightarrow (\alpha(F_1) \vee \alpha(F_2))$$

□

To be able to show the abstraction by omission preserves the k-unrolling, it will first be shown that clock constraints are preserved, that means for a clock constraint φ, $\alpha(\varphi)$ is a valid clock constraint in the sense of definition 2.4.1.

Remark 4.2.9 (Abstraction by Omission preserves Clock Constraints)
Let $\varphi \in \Phi(X)$ be a clock constraint over a set of clocks X, as obtained from definition 2.4.1, let $\mathcal{AS} \subseteq X$ be an abstraction set containing only clocks.

The abstraction function α preserves the clock constraints, that means the abstraction of φ from $\mathcal{AS}$ by omission, denoted by $\alpha(\varphi)_{-\mathcal{AS}}$, also is a well-defined clock constraint over X.

Proof. By definition, α only changes literals (see definition 4.2.1) and thus preserves the logical structure[6]. Furthermore, a literal may only be converted to `true`. Thus, by definition of clock constraints (definition 2.4.1):

$$\alpha(\varphi)_{-\mathcal{AS}} \in \Phi(X).$$

□

Using this remark, it is now possible to show the abstraction by omission preserves the formula representation and k-unrolling, respectively.

Proposition 4.2.10 (Abstraction by Omission preserves Formula Representation)
Let $\mathcal{A} = (\underline{A}, S, s_0, X, I, E)$ be a timed automaton, using the notation of remark 3.3.1. Let $\varphi(\mathcal{A})$ and $\varphi(\mathcal{A})_k$ be the formula representation and k-unrolling of $\mathcal{A}$, respectively. Remember they are in negation normal form (see remark 3.3.14). For $\mathcal{AS} \subseteq X$ being an abstraction set containing only clocks of $\mathcal{A}$, the abstraction by omission preserves the formula representation and k-unrolling, respectively; that means there exists an automaton $\mathcal{A}'$ with formula representation $\varphi(\mathcal{A}')$ and k-unrolling $\varphi(\mathcal{A}')_k$, respectively, such that

$$\varphi(\mathcal{A}') = \alpha(\varphi(\mathcal{A})) \text{ and}$$
$$\varphi(\mathcal{A}')_k = \alpha(\varphi(\mathcal{A})_k).$$

Proof. Let $\widetilde{\mathcal{A}} = (\underline{\widetilde{A}}, \widetilde{S}, \widetilde{s_0}, \widetilde{X}, \widetilde{I}, \widetilde{E})$ be a timed automaton with

- $\underline{\widetilde{A}} = \underline{A}$,
- $\widetilde{S} = S$,
- $\widetilde{s_0} = s_0$,
- $\widetilde{X} = X \setminus \mathcal{AS}$,
- $\widetilde{I}(s) = \alpha(I(s))_{-\mathcal{AS}}$ and
- $\widetilde{E} = \{(s, \alpha(\varphi)_{-\mathcal{AS}}, Y, s') \mid \text{if there exists a transition } e = (s, \varphi, Y, s') \in E\}$

Using this, $\widetilde{\mathcal{A}}$ is equal to $\mathcal{A}$ apart from the fact that the abstraction function α—with abstraction set $\mathcal{AS} \subseteq X$ as defined above—has been applied to all its clock constraints φ, that means to guards and invariants. The clock constraints are well-defined according to remark 4.2.9. Let $\varphi(\widetilde{\mathcal{A}})$ and $\varphi(\widetilde{\mathcal{A}})_k$ be the formula representation and k-unrolling of $\widetilde{\mathcal{A}}$, respectively.

The formula representation $\varphi(\mathcal{A})$ is defined as (see (3.15) in definition 3.3.10)

$$\varphi(\mathcal{A}) = \varphi_{gc}(\mathcal{A}) \wedge \varphi_E(\mathcal{A}) \wedge \varphi_i(\mathcal{A}),$$

[6]The logical structure of a formula is the order of its literals and the logical operators $\wedge$, $\vee$ and $\neg$. For example, for a formula $F = (p \vee \neg q) \wedge \neg(r \wedge \neg(x = 5))$, with $p, q, r \in \mathcal{P}$ being atomic propositions and $x \in \mathcal{V}$ being a variable, the logical structure is $F = (l_1 \vee l_2) \wedge \neg(l_3 \wedge l_4)$ (for literals l_i). Note that an occurrence of $\neg$ is part of the logical structure only if it is not part of a literal.

and the abstraction by omission of this formula is

$$\begin{aligned}\alpha(\varphi(\mathcal{A})) &= \alpha\left(\varphi_{gc}(\mathcal{A}) \wedge \varphi_E(\mathcal{A}) \wedge \varphi_i(\mathcal{A})\right) \\ &\overset{\text{def. } 4.2.1}{=} \alpha\left(\varphi_{gc}(\mathcal{A})\right) \wedge \alpha\left(\varphi_E(\mathcal{A})\right) \wedge \alpha\left(\varphi_i(\mathcal{A})\right).\end{aligned}$$

Consider the parts separately:

- The formula representation of the general constraints (that means mutual state exclusion) does not contain clocks (but only states, see definition 3.3.9), that means $\mathcal{AS} \cap \Sigma(\varphi_{gc}(\mathcal{A})) = \emptyset$, and thus applying α does not change the formula (see also definition 4.2.1):

  $$\alpha(\varphi_{gc}(\mathcal{A})) = \varphi_{gc}(\mathcal{A}).$$

  Considering definition 3.3.9, $\alpha(\varphi_{gc}(\mathcal{A}))$ obviously is a well-defined general constraint representation. Furthermore,

  $$\alpha(\varphi_{gc}(\mathcal{A})) = \varphi_{gc}(\widetilde{\mathcal{A}}) \tag{4.2}$$

  (directly follows from definition of $\widetilde{\mathcal{A}}$), as the set of states of automaton $\mathcal{A}$ and automaton $\widetilde{\mathcal{A}}$ is identical.

- The abstraction of the transition relation is defined as:

  $$\begin{aligned}\alpha(\varphi_E(\mathcal{A})) &\overset{\text{def. } 3.3.7}{=} \alpha\left(\bigvee_{(s,\varphi,Y,s') \in E} e_a((s,\varphi,Y,s')) \vee \bigvee_{s \in S} e_d(s)\right) \\ &\overset{\text{def. } 4.2.1}{=} \bigvee_{(s,\varphi,Y,s') \in E} \alpha\left(e_a((s,\varphi,Y,s'))\right) \vee \bigvee_{s \in S} \alpha(e_d(s))\end{aligned}$$

  Consider a single transition (s, φ, Y, s'), together with its formula representation $e_a = ((s, \varphi, Y, s'))$:

  $$\begin{aligned}\alpha\left(e_a((s,\varphi,Y,s'))\right) &\overset{\text{def. } 3.3.5}{=} \alpha\Bigg(\overline{\mathtt{s}}_t \wedge \overline{\mathtt{s}}'_{t+1} \wedge \varphi_t \wedge (\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1}) \wedge \\ &\qquad \bigwedge_{x \in Y} (\overline{\mathtt{x}}_{t+1} = \overline{\mathtt{z}}_{t+1}) \wedge \bigwedge_{x \notin Y} (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1})\Bigg) \\ &\overset{\text{def. } 4.2.1}{=} \alpha(\overline{\mathtt{s}}_t) \wedge \alpha(\overline{\mathtt{s}}'_{t+1}) \wedge \alpha(\varphi_t) \wedge \alpha(\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1}) \wedge \\ &\qquad \bigwedge_{x \in Y} \alpha\left(\overline{\mathtt{x}}_{t+1} = \overline{\mathtt{z}}_{t+1}\right) \wedge \bigwedge_{x \notin Y} \alpha(\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \\ &\overset{s,s',z \notin \mathcal{AS}}{=} \overline{\mathtt{s}}_t \wedge \overline{\mathtt{s}}'_{t+1} \wedge \widetilde{\varphi}_t \wedge (\overline{\mathtt{z}}_t = \overline{\mathtt{z}}_{t+1}) \wedge \\ &\qquad \bigwedge_{x \in Y \setminus \mathcal{AS}} (\overline{\mathtt{x}}_{t+1} = \overline{\mathtt{z}}_{t+1}) \wedge \bigwedge_{x \in Y \setminus \mathcal{AS}} (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}),\end{aligned}$$

with $\alpha(\varphi_t) = \widetilde{\varphi}_t$. $\widetilde{\varphi}_t$ is a well-defined guard according to remark 4.2.9. Considering definition 3.3.5, $\alpha(e_a((s, \varphi, Y, s')))$ is a well-defined representation of an action transition. Furthermore, from the above follows

$$\alpha(e_a((s, \varphi, Y, s'))) = e_a((s, \alpha(\varphi), Y, s')).$$

By definition of $\widetilde{E}$, for (s, φ, Y, s') being a transition of $\mathcal{A}$, $(s, \alpha(\varphi), Y, s')$ is a transition of $\widetilde{\mathcal{A}}$, and as this holds for every transition (s, φ, Y, s'):

$$\bigvee_{(s,\varphi,Y,s')\in E} \alpha\left(e_a((s, \varphi, Y, s'))\right) = \bigvee_{(s,\alpha(\varphi),Y,s')\in \widetilde{E}} e_a((s, \alpha(\varphi), Y, s')). \tag{4.3}$$

Now, consider the formula representation $e_d(s)$ for a single delay step for a certain state s:

$$\begin{aligned}
\alpha(e_d(s)) &\overset{\text{def. 3.3.6}}{=} \alpha\Bigg(\overline{\mathtt{s}}_t \wedge \overline{\mathtt{s}}_{t+1} \wedge I_t(\overline{\mathtt{s}}_t) \wedge I_{t+1}(\overline{\mathtt{s}}_{t+1}) \wedge \\
&\qquad \bigwedge_{x\in X} (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{z}}_t < \overline{\mathtt{z}}_{t+1})\Bigg) \\
&\overset{\text{def. 4.2.1}}{=} \alpha(\overline{\mathtt{s}}_t) \wedge \alpha(\overline{\mathtt{s}}_{t+1}) \wedge \alpha(I_t(\overline{\mathtt{s}}_t)) \wedge \alpha(I_{t+1}(\overline{\mathtt{s}}_{t+1})) \wedge \\
&\qquad \bigwedge_{x\in X} \alpha(\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge \alpha(\overline{\mathtt{z}}_t < \overline{\mathtt{z}}_{t+1}) \\
&\overset{s,z\notin\mathcal{AS}}{=} \overline{\mathtt{s}}_t \wedge \overline{\mathtt{s}}_{t+1} \wedge \widetilde{I}_t(\overline{\mathtt{s}}_t) \wedge \widetilde{I}_{t+1}(\overline{\mathtt{s}}_{t+1}) \wedge \\
&\qquad \bigwedge_{x\in X\setminus\mathcal{AS}} (\overline{\mathtt{x}}_t = \overline{\mathtt{x}}_{t+1}) \wedge (\overline{\mathtt{z}}_t < \overline{\mathtt{z}}_{t+1}),
\end{aligned}$$

with $\alpha(I_t(\overline{\mathtt{s}}_t)) = \widetilde{I}_t(\overline{\mathtt{s}}_t)$ and $\alpha(I_{t+1}(\overline{\mathtt{s}}_{t+1})) = \widetilde{I}_{t+1}(\overline{\mathtt{s}}_{t+1})$. $\widetilde{I}_t(\overline{\mathtt{s}}_t)$ and $\widetilde{I}_{t+1}(\overline{\mathtt{s}}_{t+1})$ are well-defined invariants according to remark 4.2.9. Considering definition 3.3.6, $\alpha(e_d(s))$ is a well-defined representation of a delay step. Moreover, using the definition of $\widetilde{I}$, from the above follows

$$\alpha(e_d(s)) = e_d(\widetilde{s})$$

for $s \in S, \widetilde{s} \in \widetilde{S}$, and thus

$$\bigvee_{s\in S} \alpha(e_d(s)) = \bigvee_{s\in\widetilde{S}} e_d(s). \tag{4.4}$$

Considering definition 3.3.7, from (4.3) and (4.4) follows:

$$\alpha(\varphi_E(\mathcal{A})) = \varphi_E(\widetilde{\mathcal{A}}) \tag{4.5}$$

- For the initial conditions:

$$\begin{aligned}\alpha\left(\varphi_i(\mathcal{A})\right) &\overset{\text{def. 3.3.8}}{=} \alpha(\overline{\mathtt{s\_0}}_0 \wedge (\overline{\mathtt{z}}_0 = 0) \wedge \bigwedge_{x \in X} (\overline{\mathtt{x}}_0 = 0)) \\ &\overset{def.4.2.1}{=} \alpha(\overline{\mathtt{s\_0}}_0) \wedge \alpha(\overline{\mathtt{z}}_0 = 0) \wedge \bigwedge_{x \in X} \alpha((\overline{\mathtt{x}}_0 = 0)) \\ &\overset{s_0, z \notin \mathcal{AS}}{=} \overline{\mathtt{s\_0}}_0 \wedge (\overline{\mathtt{z}}_0 = 0) \wedge \bigwedge_{x \in X \setminus \mathcal{AS}} (\overline{\mathtt{x}}_0 = 0)\end{aligned}$$

  Thus, $\alpha(\varphi_i(\mathcal{A}))$ is a well-defined representation of initial constraints as defined in definition 3.3.8, and

$$\alpha(\varphi_i(\mathcal{A})) = \varphi_i(\widetilde{\mathcal{A}}) \tag{4.6}$$

  (directly follows from definition of $\widetilde{\mathcal{A}}$).

Consider again $\alpha(\varphi(\mathcal{A}))$:

$$\begin{aligned}\alpha(\varphi(\mathcal{A})) &= \alpha\left(\varphi_{gc}(\mathcal{A})\right) \wedge \alpha\left(\varphi_E(\mathcal{A})\right) \wedge \alpha\left(\varphi_i(\mathcal{A})\right) \overset{(4.2),(4.5),(4.6)}{=} \\ &= \varphi_{gc}(\widetilde{\mathcal{A}}) \wedge \varphi_E(\widetilde{\mathcal{A}}) \wedge \varphi_i(\widetilde{\mathcal{A}}) \overset{\text{def.}\ \widetilde{\mathcal{A}}}{=} \\ &= \varphi(\widetilde{\mathcal{A}}),\end{aligned}$$

that means applying the abstraction function α to the formula representation $\varphi(\mathcal{A})$ of $\mathcal{A}$ returns the formula representation $\varphi(\widetilde{\mathcal{A}})$ of automaton $\widetilde{\mathcal{A}}$.

The argumentation for

$$\varphi(\widetilde{\mathcal{A}})_k = \alpha(\varphi(\mathcal{A})_k)$$

is done in the same style.

Thus, the abstraction by omission preserves the formula representation and k-unrolling. □

Using this proposition, the "middle part" of figure 4.2 is shown to be a partially commuting diagram. Partially commuting in this context means: For a k-unrolling $\varphi(\mathcal{A})_k$ of a timed automaton $\mathcal{A}$ and an interpretation $\sigma \in \mathit{Mod}(\mathit{mcci})$ being model for $\varphi(\mathcal{A})_k$, σ is also an interpretation for $\alpha(\varphi(\mathcal{A})_k)$. But for $\sigma' \in \mathit{Mod}(\mathit{mcci})$ being model for $\alpha(\varphi(\mathcal{A})_k)$, σ' is not necessarily a model for $\varphi(\mathcal{A})_k$. This fact is expressed by the "$\subseteq$" sign labelling the lower middle part of figure 4.2.

Remark 4.2.11 (Commuting Subdiagram)
The subpart of figure 4.2 depicted in figure 4.5 is a partially commuting diagram.

Proof. Let $\varphi(\mathcal{A})_k$ be the k-unrolling for some automaton $\mathcal{A}$, let $\alpha(\varphi(\mathcal{A})_k)$ be the abstraction of $\varphi(\mathcal{A})_k$ by omission.

The abstraction function α preserves the k-unrolling (see proposition 4.2.10), that means $\alpha(\varphi(\mathcal{A})_k)$ also is the k-unrolling of some other automaton $\widetilde{\mathcal{A}}$ (upper part of figure 4.5).

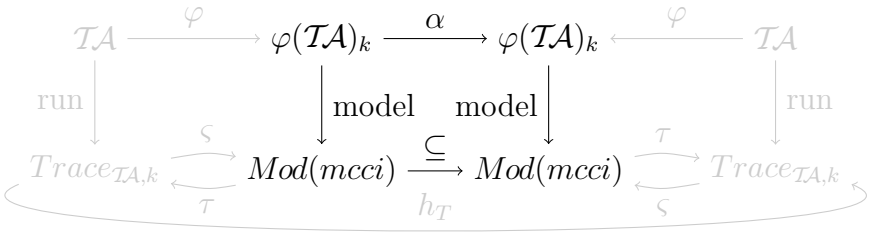


Figure 4.5.: Abstraction by Omission: Second Commuting Subdiagram

As has been shown in lemma 4.2.8, the abstraction by omission is a conservative approximation, that means

$$\models \varphi(\mathcal{A})_k \rightarrow \alpha(\varphi(\mathcal{A})_k)$$

($\varphi(\mathcal{A})_k \rightarrow \alpha(\varphi(\mathcal{A})_k)$ is a valid formula, see also page 13). Thus, all interpretations $\sigma \in Mod(mcci)$ are a model for $\varphi(\mathcal{A})_k \rightarrow \alpha(\varphi(\mathcal{A})_k)$, that means

$$\mathcal{I}[\![\varphi(\mathcal{A})_k \rightarrow \alpha(\varphi(\mathcal{A})_k)]\!](\sigma) = \mathtt{tt}.$$

Let $\sigma' \in Mod(mcci)$ be a model for $\varphi(\mathcal{A})_k$ (left part of figure 4.5), that means $\mathcal{I}[\![\varphi(\mathcal{A})_k]\!](\sigma) = \mathtt{tt}$ (see also definition 2.1.10). As $\varphi(\mathcal{A})_k \rightarrow \alpha(\varphi(\mathcal{A})_k)$ is valid, σ' also is a model for this formula: $\mathcal{I}[\![\varphi(\mathcal{A})_k \rightarrow \alpha(\varphi(\mathcal{A})_k)]\!](\sigma') = \mathtt{tt}$.

Thus, by definition of $\rightarrow$ and the semantics of formulae, σ' also is a model for $\alpha(\varphi(\mathcal{A})_k)$, that means

$$\mathcal{I}[\![\alpha(\varphi(\mathcal{A})_k)]\!](\sigma') = \mathtt{tt}$$

and hence, figure 4.5 is a partially commuting diagram. □

What remains to be shown for figure 4.2 to be a commuting diagram is the existence of a homomorphism h_T, as defined in definition 4.2.2.

Proof of theorem 4.2.6. Let $\mathcal{A}$ be timed automaton, let $\varphi(\mathcal{A})_k$ be its k-unrolling for some $k \in \mathbb{N}$, let $Mod(\varphi(\mathcal{A})_k)$ be the set of interpretations being model for $\varphi(\mathcal{A})_k$, let $Trace_{\mathcal{A},k}$ be the set of finite traces of length k of $\mathcal{A}$, and let $\mathcal{AS} \subseteq X$ be the abstraction set containing only clocks from $\mathcal{A}$. Let $\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}$ be the abstraction from $\varphi(\mathcal{A})_k$ by omission, let $Mod(\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}})$ be the set of interpretations being model for $\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}$. For every interpretation $\sigma \in Mod(\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}})$, let t_σ be the corresponding trace (see definition 3.3.21), let $Trace_{\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}}$ be the set of all these traces.

For a trace $t \in Trace_{\mathcal{A},k}$,

$$t : (s_0, \nu_0) \xrightarrow{\alpha_0} (s_1, \nu_1) \xrightarrow{\alpha_1} (s_2, \nu_2) \ldots \xrightarrow{\alpha_{k-1}} (s_k, \nu_k),$$

let $\varsigma(t) = \sigma_t \in Mod(\varphi(\mathcal{A})_k)$ be the corresponding interpretation of t (see definition 3.3.18), let $\sigma' \in Mod(\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}})$ be an interpretation defined as

$$\sigma'(v) = \begin{cases} \sigma_t(v), & v \in \alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}} \\ \text{undefined}, & \text{otherwise} \end{cases}$$

for $v \in \mathcal{V} \cup \mathcal{P}$[7] (see also remark 4.2.11). Note that σ' is well-defined, as by definition of α: $Var(\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}) \subseteq Var(\varphi(\mathcal{A})_k)$. Let $\tau(\sigma') = t_{\sigma'} \in Trace_{\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}}$ be the corresponding trace of σ' (see definition 3.3.21),

$$t_{\sigma'} : (\widetilde{s_0}, \widetilde{\nu_0}) \xrightarrow{\widetilde{\alpha_0}} (\widetilde{s_1}, \widetilde{\nu_1}) \xrightarrow{\widetilde{\alpha_1}} (\widetilde{s_2}, \widetilde{\nu_2}) \dots \xrightarrow{\widetilde{\alpha_{k-1}}} (\widetilde{s_k}, \widetilde{\nu_k}).$$

Let $h_T : Trace_{\mathcal{A},k} \to Trace_{\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}}$ be a mapping, with $h_T(t) = t_{\sigma'}$ for traces $t, t_{\sigma'}$ as defined above. Using this, h_T is a homomorphism as defined in definition 4.2.2:

- By definition of α, $\varphi(\mathcal{A})_k$ and $\alpha(\varphi(\mathcal{A})_k)$ contain the same set of actions A. From remark 4.2.11 follows: $\sigma'(\overline{\alpha}) = \sigma_t(\overline{\alpha})$, for all actions $\alpha \in A$ and $\overline{\alpha}$ being the representation of α. Thus, by definition of corresponding interpretation and corresponding trace: $\alpha_i = \widetilde{\alpha_i}$, $i \in \{0, \dots, k-1\}$, for all transitions $(s_i, \nu_i) \xrightarrow{\alpha_i} (s_{i+1}, \nu_{i+1})$ and $(\widetilde{s_i}, \widetilde{\nu_i}) \xrightarrow{\widetilde{\alpha_i}} (\widetilde{s_{i+1}}, \widetilde{\nu_{i+1}})$ as defined above, that means the sequences of labels of t and $t_{\sigma'}$ are identical.

- Let $X = \mathcal{AS} \dot{\cup} \overline{\mathcal{AS}}$. By definition of α and $Trace_{\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}}$, traces contained in $Trace_{\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}}$ only contain clocks from $\overline{\mathcal{AS}}$. For a valuation ν_i (obtained from t) and a clock x in $\overline{\mathcal{AS}}$, let $\overline{\mathtt{x}}_i$ be the representation of x "at step i", as defined in definition 3.3.18. By definition of σ': $\sigma'(\overline{\mathtt{x}}_i) = \sigma_t(\overline{\mathtt{x}}_i)$, and hence by definition of corresponding trace (definition 3.3.21): $\nu_i(\overline{\mathtt{x}}_i) = \widetilde{\nu_i}(\overline{\mathtt{x}}_i)$.

  Finally, h_T is a homomorphism as defined in definition 4.2.2. Using the results of remark 4.2.7, proposition 4.2.10 and remark 4.2.11, figure 4.2 is a commuting diagram and thus, the abstraction by omission is correct.

□

Improved Abstraction

The abstraction function α, as presented in definition 4.2.1, is a correct for abstraction of clocks, as has been shown in the previous section. However, α is not efficient, as formulae containing `true` as a subformulae may be simplified in most cases. Therefor, an improved version α_s (still working on formulae in negation normal form) of the abstraction function α will be presented.

The abstraction function α works top-down, in that it is recursively applied to subformulae (see figure 4.1). In contrast to that, α_s works bottom-up. The basic idea of α_s is the fact that a formula containing `true` as subformula may be simplified in consideration of the context (that means in consideration of the logical structure) the subformula appears in.

Definition 4.2.12 (Abstraction by Omission, Improved Form)
Let $\mathcal{A}$ be a timed automaton, let $\varphi(\mathcal{A})$ and $\varphi(\mathcal{A})_k$ be its formula representation and k-unrolling for some $k \in \mathbb{N}$, respectively. Let both $\varphi(\mathcal{A})$ and $\varphi(\mathcal{A})_k$ be in negation normal form (see remark 3.3.14), let $\mathcal{AS}$ be the abstraction set.

[7] Roughly speaking, σ' is obtained from restricting σ_t to $\alpha(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}$.

The *improved abstraction of* $\varphi(\mathcal{A})$ *from* $\mathcal{AS}$ *by omission*, denoted by $\alpha_s(\varphi(\mathcal{A}))$, is defined by applying transformation function α_s (depicted in figure 4.6) to $\varphi(\mathcal{A})$. The *improved abstraction of* $\varphi(\mathcal{A})_k$ *from* $\mathcal{AS}$ *by omission*, denoted by $\alpha_s(\varphi(\mathcal{A})_k)$, is defined in the same way.

To avoid ambiguities regarding the abstraction set, $\alpha_s(\varphi(\mathcal{A}))$ and $\alpha_s(\varphi(\mathcal{A})_k)$ may also be written as $\alpha_i(\varphi(\mathcal{A}))_{-\mathcal{AS}}$ and $\alpha_i(\varphi(\mathcal{A})_k)_{-\mathcal{AS}}$, respectively. □

$$\alpha_s : \mathcal{F}(\mathcal{P}, \mathcal{V})_{nnf} \to \mathcal{F}(\mathcal{P} \setminus \mathcal{AS}, \mathcal{V} \setminus \mathcal{AS})_{nnf}$$

$$\alpha_s(L) = \begin{cases} L & \Leftarrow \Sigma(L) \cap \mathcal{AS} = \emptyset \\ \texttt{true} & \Leftarrow \Sigma(L) \cap \mathcal{AS} \neq \emptyset \end{cases}$$

$$\alpha_s(F \wedge G) = \begin{cases} \alpha_s(F) \wedge \alpha_s(G) & \Leftarrow \alpha_s(F) \neq \texttt{true} \text{ and } \alpha_s(G) \neq \texttt{true} \\ \alpha_s(F) & \Leftarrow \alpha_s(G) = \texttt{true} \text{ and } \alpha_s(F) \neq \texttt{true} \\ \alpha_s(G) & \Leftarrow \alpha_s(F) = \texttt{true} \text{ and } \alpha_s(G) \neq \texttt{true} \\ \texttt{true} & \Leftarrow \alpha_s(F) = \alpha_s(G) = \texttt{true} \end{cases}$$

$$\alpha_s(F \vee G) = \begin{cases} \alpha_s(F) \vee \alpha_s(G) & \Leftarrow \alpha_s(F) \neq \texttt{true} \text{ and } \alpha_s(G) \neq \texttt{true} \\ \texttt{true} & \Leftarrow \alpha_s(F) = \texttt{true} \text{ or } \alpha_s(G) = \texttt{true} \end{cases}$$

Let F and G be formulae in negation normal form, let L be a literal (see page 8), let $\Sigma(L)$ be the corresponding signature (that means the set of variables appearing in L, see definition 2.1.11). Note that $\alpha_s(F) = F$ in case $\mathcal{AS} \cap \Sigma(F) = \emptyset$, that means if a formula F does not contain variables appearing in the abstraction set, the improved abstraction function α_s returns the identity.

Figure 4.6.: Abstraction by Omission, Improved Form

Note that for a formula F, $\alpha_s(F)$ will be smaller than $\alpha(F)$, that means the absolute number of literals and logical symbols in $\alpha_s(F)$ is smaller (or at most equal) than the absolute number of literals and logical symbols in $\alpha(F)$. This is due to the fact that α only replaces literals with `true`, while α_s omits them, and in fact, α_s not only omits literals but also larger subformulae. Nevertheless, $\alpha(F)$ and $\alpha_s(F)$ are semantically equivalent, as will be shown by the following remark.

Remark 4.2.13 (Equivalence of α and α_s)
Let F be a formula in negation normal form, let $\mathcal{AS}$ be the abstraction set, let $\alpha(F)$ be the abstraction of F by omission (according to definition 4.2.1), and let α_s (F) be the improved abstraction of F by omission (according to definition 4.2.12). $\alpha(F)$ and α_s (F) are semantically equivalent, that means

$$\models \alpha(F) \leftrightarrow \alpha_s(F)$$

Proof. Let L be a literal. The proof is done inductively on the structure of F:

IA: $F = L$: As the definitions of α and α_s agree on literals (see figures 4.1 and 4.6), $(\models \alpha(L) \leftrightarrow \alpha_s(L))$ holds trivially.

IH: For formulae F_1 and F_2 being subformulae of F, let $(\models \alpha(F_1) \leftrightarrow \alpha_s(F_1))$ and $(\models \alpha(F_2) \leftrightarrow \alpha_s(F_2))$.

IS: $F = F_1 \wedge F_2$: By IH and propositional logic (definition of $\leftrightarrow$ and $\wedge$):

$$\models (\alpha(F_1) \wedge \alpha(F_2)) \leftrightarrow (\alpha_s(F_1) \wedge \alpha_s(F_2)), \tag{*}$$

and, by definition 4.2.1:

$$\alpha(F_1 \wedge F_2) = \alpha(F_1) \wedge \alpha(F_2) \tag{**}$$

- If $\alpha_s(F_1) \neq \texttt{true}$ and $\alpha_s(F_2) \neq \texttt{true}$, then $\alpha_s(F_1 \wedge F_2) \overset{\text{def. } 4.2.12}{=} \alpha_s(F_1) \wedge \alpha_s(F_2)$. Thus,

  $$\models \alpha(F_1) \wedge \alpha(F_2) \leftrightarrow \alpha_s(F_1) \wedge \alpha_s(F_2)$$

  directly follows from (*) and (**).

- If $\alpha_s(F_1) \neq \texttt{true}$ and $\alpha_s(F_2) = \texttt{true}$, then $\alpha_s(F_1 \wedge F_2) \overset{\text{def. } 4.2.12}{=} \alpha_s(F_1)$. $\models (\alpha(F_1) \wedge \alpha(F_2)) \leftrightarrow (\alpha_s(F_1) \wedge \texttt{true})$ by (*), and $\models (\alpha_s(F_1) \wedge \texttt{true}) \leftrightarrow \alpha_s(F_1)$ by definition of $\wedge$. Thus, by (**):

  $$\models \alpha(F_1) \wedge \alpha(F_2) \leftrightarrow \alpha_s(F_1).$$

- If $\alpha_s(F_1) = \texttt{true}$ and $\alpha_s(F_2) \neq \texttt{true}$, then $\alpha_s(F_1 \wedge F_2) \overset{\text{def. } 4.2.12}{=} \alpha_s(F_2)$. $\models (\alpha(F_1) \wedge \alpha(F_2)) \leftrightarrow (\texttt{true} \wedge \alpha_s(F_2))$ by (*), and $\models (\texttt{true} \wedge \alpha_s(F_2)) \leftrightarrow \alpha_s(F_2)$ by definition of $\wedge$. Thus, by (**):

  $$\models \alpha(F_1) \wedge \alpha(F_2) \leftrightarrow \alpha_s(F_2).$$

- If $\alpha_s(F_1) = \texttt{true}$ and $\alpha_s(F_2) = \texttt{true}$, then $\alpha_s(F_1 \wedge F_2) \overset{\text{def. } 4.2.12}{=} \texttt{true}$ $\models (\alpha(F_1) \wedge \alpha(F_2)) \leftrightarrow \texttt{true} \wedge \texttt{true})$ by (*), and $\models (\texttt{true} \wedge \texttt{true}) \leftrightarrow \texttt{true}$ by definition of $\wedge$. Thus, by (**):

  $$\models \alpha(F_1) \wedge \alpha(F_2) \leftrightarrow \alpha_s(F_1).$$

$F = F_1 \vee F_2$: By IH and propositional logic (definition of $\leftrightarrow$ and $\vee$):

$$\models (\alpha(F_1) \vee \alpha(F_2)) \leftrightarrow (\alpha_s(F_1) \vee \alpha_s(F_2)), \tag{$\star$}$$

and, by definition 4.2.1,

$$\alpha(F_1 \vee F_2) = \alpha(F_1) \vee \alpha(F_2). \tag{$\star\star$}$$

- If $\alpha_s(F_1) \neq \texttt{true}$ and $\alpha_s(F_2) \neq \texttt{true}$, then $\alpha_s(F_1 \vee F_2) \overset{\text{def. } 4.2.12}{=} \alpha_s(F_1) \vee \alpha_s(F_2)$. Thus,

  $$\models \alpha(F_1 \vee F_2) \leftrightarrow \alpha_s(F_1) \vee \alpha_s(F_2)$$

  directly follows from ($\star$) and ($\star\star$).

– If $\alpha_s(F_1) = \texttt{true}$ or $\alpha_s(F_2) = \texttt{true}$, then $\alpha_s(F_1 \vee F_2) \overset{\text{def. } 4.2.12}{=} \texttt{true}$. $\models (\alpha(F_1) \vee \alpha(F_2)) \leftrightarrow (\texttt{true} \vee \texttt{true})$ by $(\star)$, and $\models \texttt{true} \vee \texttt{true} \leftrightarrow \texttt{true}$ by definition of $\vee$. Thus, by $(\star\star)$,

$$\models (\alpha(F_1) \vee \alpha(F_2)) \leftrightarrow \texttt{true}.$$

□

Thus, for a formula F, the formulae $\alpha(F)$ and $\alpha_s(F)$ are semantically equivalent, and all results from the preceeding section obtained for α also hold for α_s. This means in particular: α_s is correct. Therefor, when talking about abstraction by omission in the sequel, this shall refer to the improved version α_s.

Conclusion

The abstraction by omission is a correct abstraction for "omissible parameters", but it is not sensible for arbitrary parameters, as the resulting formulae might not be well-defined (in the sense of section 3.3) any more. For a timed automaton $\mathcal{A}$, omissible parameters are those which can be removed from $\mathcal{A}$ without loosing well-definedness, that means the result after removing the parameters still is a well-defined timed automaton according to definition 2.4.3. Examples for omissible parameters are clocks, clock constraints (guards as well as invariants) and events.

For omissible parameters, abstraction by omission is a simple but powerful abstraction technique: The abstraction function is very concise and can easily be applied to formulae. Furthermore, with respect to reachability analysis, abstraction by omission will yield results very quickly, as omissible parameters are those which mainly restrict reachability (see also page 67).

On the other hand, abstraction by omission is not reasonable for states, as removing a state variable s without paying attention to the constraints associated with it does not preserve the formula representation. To abstract from states, further methods to also remove the invariant and to somehow preserve the transition relation would be needed in addition to the abstraction by omission function. Possible approaches for state abstraction are presented in section 4.2.2.

4.2.2. Abstract from States

The abstraction of states may be done according to (Dierks, 1999): Two states s and s' are combined to a new state $\widetilde{s}$ by

1. redirecting all in- and outgoing transitions of s and s' to $\widetilde{s}$
2. creating an invariant for $\widetilde{s}$ by disjunction of the invariants of s and s'.

5. Craig Interpolation

5.1. Overview

Craig interpolants(Craig, 1957b) are used in model checking and abstraction refinement in a variety of cases, consider for example (McMillan & Jhala, 2005), (McMillan, 2005a), (McMillan, 2003) and (Henzinger *et al.*, 2004).

Definition 5.1.1 (Craig Interpolant)
Let (A, B) be a pair of formulae, with $\not\models A \wedge B$. A *Craig interpolant for A and B* is a formula C such that

$$\models A \rightarrow C \tag{5.1}$$
$$\models \neg(C \wedge B) \tag{5.2}$$
$$\Sigma(C) \subseteq \Sigma(A) \cap \Sigma(B) \tag{5.3}$$

Thus, a Craig interpolant for an inconsistent pair of formulae (A, B) is a formula that is implied by A, inconsistent with B and refers only to the common variables of A and B. For C being the interpolant of A and B, A is called *prefix of C* and B is called *suffix of C*.

As Craig interpolants are the only interpolants considered in this thesis, they will simply be called interpolants. □

Remark 5.1.2 (Interpolant for a Pair of Formula Sets)
Craig interpolants are not only defined for a pair of formulae (A, B), but also for a pair of sets of formulae, that means for $(\{A_1, \ldots, A_n\}, \{B_1, \ldots, B_m\})$. For C being an interpolant for the pair of sets, the intended meaning is that C is an interpolant for the conjunction $(A_1 \wedge \ldots \wedge A_n, B_1 \wedge \ldots \wedge B_m)$ (see also remark 2.1.4). □

Note that in general, interpolants are not unique. This follows directly from definition 5.1.1. However, there always exists a weakest and a strongest interpolant:

Definition 5.1.3 (Weakest and strongest interpolant)
Let (A, B) be a pair of formulae. The *weakest interpolant* for (A, B), is the interpolant C that is implied by all other interpolants for (A, B). The *strongest interpolant* for (A, B), is the interpolant C that implies all other interpolants for (A, B).

The notation of *the* weakest and *the* strongest interpolant is well-defined, as weakest and strongest interpolant are unique. This has been proven in (Kiefer, 2005), for example. □

By definition, an interpolant C for a pair (A, B) always is an over-approximation of the set of valuations satisfying the prefix (see (5.1)). Equivalently, the interpolant

always is an under-approximation of the set of valuation which are inconsistent with the suffix (see (5.2)). Thus, the interpolant is an abstraction of the prefix containing only the information which is relevant for the inconsistency of prefix and suffix and thus is a way of filtering out irrelevant information from the prefix.

For formulae A and B and a resolution proof of unsatisfiability, an interpolant for (A, B) can be derived in linear time, which will be shown in the next section. When using a resolution proof obtained from a SAT solver, the interpolant may be seen as a way of capturing the facts which the solver derived about the prefix and which are relevant for the inconsistence.

5.2. Deriving Interpolants from Proofs

The interpolation theorem shown in (Craig, 1957b) states that for an inconsistent pair of formulae, an interpolant always exists. However, to be able to use them for model checking, a proof of existence is not enough, the interpolants also have to be effectively computable.

In (Pudlák, 1997), (McMillan, 2003) and (McMillan, 2004), algorithms and rules are presented to derive an interpolant for an inconsistent pair of formulae from a resolution proof. To provide a short insight into the technique of deriving interpolants and how interpolants in an interpolant sequence are related, the algorithm of (McMillan, 2003) (which, for propositional formulae, equals the one in (McMillan, 2004)) will be briefly presented.

Conceptually, the algorithm works as follows: Given a resolution proof of unsatisfiability for two formulae A and B. Starting with the premises, every clause of the proof is annotated with an intermediate formula representing information about the clause itself (in case of a premise) or the satisfiability of the clauses where this clause has been resolved from. The annotated formulae will be called "intermediate interpolants" in the sequel. The formula annotated to the root node (which is $\square$, as it is a proof of unsatisfiability) represents the (final) interpolant for A and B.

Definition 5.2.1 (Interpolant for Resolution Proof)
Let A and B be two propositional clause sets, let $(G_1, \ldots, G_n, \square)$ be a resolution proof of $\square$ from A and B (see also definition 2.1.13), that means a proof of unsatisfiability. A propositional variable is called *local to* A if it appears in clauses of A but not in clauses of B, otherwise, it is called *global*. A literal is local to A iff the variable it contains is local to A. For c being a clause, let $g(c)$ denote the disjunction of all global literals appearing in c.

All clauses c of the proof are annotated with a formula $i(c)$ according to the following rules:

$i(c) = g(c)$,	if c is a clause from A
$i(c) = \texttt{true}$,	if c is a clause from B
$i(c) = i(c_1) \vee i(c_2)$,	if $\{c_1, c_2\} \vdash c$ is a resolution step on a variable local to A
$i(c) = i(c_1) \wedge i(c_2)$,	if $\{c_1, c_2\} \vdash c$ is a resolution step on a variable not local to A

$i(\square)$ is the interpolant for A and B.

For the proof of soundness, that means to show $i(\square)$ really is an interpolant for the pair (A, B), please consider (McMillan, 2004). $\square$

Using this definition, the complexity for deriving an interpolant from a given resolution proof of unsatisfiability is linear in the size of the proof, although the proof itself may be exponential in the size of $A \wedge B$ in the worst case[1]. Note that the interpolant for A and B derived in this way is unique with regard to the proof, but as there may be several proofs of unsatisfiability for the same clause sets, the interpolant is not unique in general.

Although definition 5.2.1 derives an interpolant for propositional formulae only, rules for dealing with other formulae may be defined in a similar way. For further details, please refer to (McMillan, 2004): The author presents rules to compute an interpolant not only for propositional formulae, but also for linear inequalities, equalities and uninterpreted function symbols[2]. The complexity of deriving interpolants for these formulae is linear in the size of the refutation proof, too, which is also shown in (McMillan, 2004).

In general, for a given clause set $\{c_1, c_2, \ldots, c_n\}$ and a proof of unsatisfiability for this set, interpolants may be derived for every possible pair of nonempty subsets, using the same resolution proof. If, however, interpolants are derived for every pair in the sequence $(\{c_1, \ldots, c_i\}, \{c_{i+1}, \ldots, c_n\})$, $i \in \{1, \ldots, n-1\}$, further correlations (in addition to those mentioned in (5.1), (5.2) and (5.3)) can be identified. Consider the following example first.

Example 5.2.2 (Deriving interpolants from resolution proof)
Consider the clause set $C = \{(d \vee f), (\neg b), (b), (\neg b \vee c), (b \vee \neg f), (\neg b \vee d), (\neg d \vee \neg e), (\neg d \vee \neg c \vee e)\}$ and the proof of unsatisfiability for C as depicted in figure 5.1. For ease of explanation, the variable used in every single resolution step is also shown in the figure.

Let

$$(\{(d \vee f), (\neg b), (b), (\neg b \vee c)\}, \{(b \vee \neg f), (\neg b \vee d), (\neg d \vee \neg e), (\neg d \vee \neg c \vee e)\})$$
$$(\{(d \vee f), (\neg b), (b), (\neg b \vee c), (b \vee \neg f)\}, \{(\neg b \vee d), (\neg d \vee \neg e), (\neg d \vee \neg c \vee e)\})$$
$$(\{(d \vee f), (\neg b), (b), (\neg b \vee c), (b \vee \neg f), (\neg b \vee d)\}, \{(\neg d \vee \neg e), (\neg d \vee \neg c \vee e)\})$$

be three possible partitions of the clause set C, denoted by (A_1, B_1), (A_2, B_2) and (A_3, B_3), respectively. Using definition 5.2.1 and the proof in figure 5.1, interpolants may be derived for every single pair, which are depicted in figure 5.2. Note that 5.2 mirrors the structure of the proof in figure 5.1, but every node c is tripartite: The first line represents the formula $i(c)$ for this node for the partition of C defined by (A_1, B_1), the second line represents $i(c)$ for (A_2, B_2) and the third line represents $i(c)$ for (A_3, B_3). Note that the formulae have been slightly simplified if possible.

The derivation will be explained for the pair (A_1, B_1) (first line in every node):

[1] When working with interpolants, most likely the resolution proof will be obtained from a SAT solver, so the complexity of finding a resolution proof does not represent a severe constraint.

[2] Remember the formula representation defined in section 3.3 contains propositional formulae as well as linear inequalities and equalities. The results of (McMillan, 2004) show that it is indeed possible to derive interpolants for the formula representation

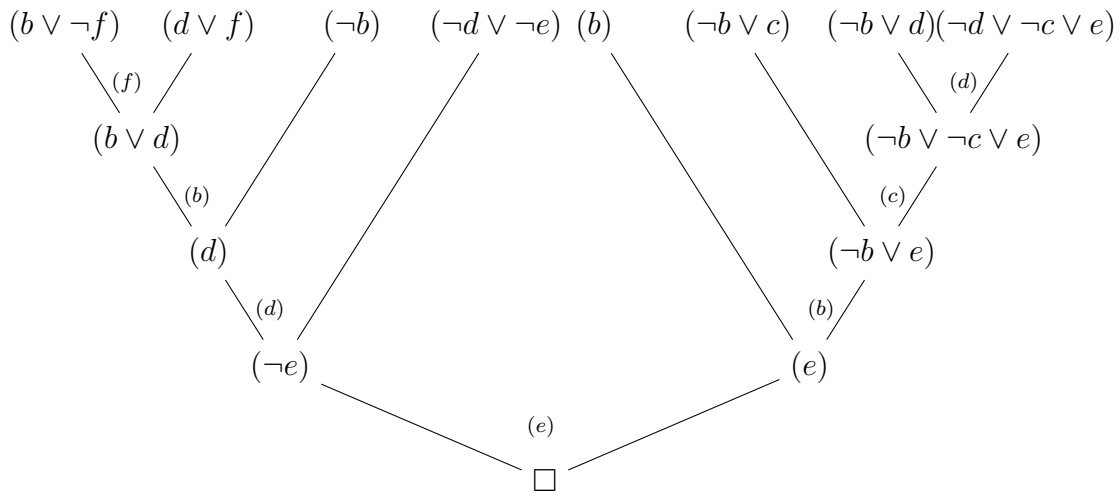


Figure 5.1.: Resolution proof of inconsistency

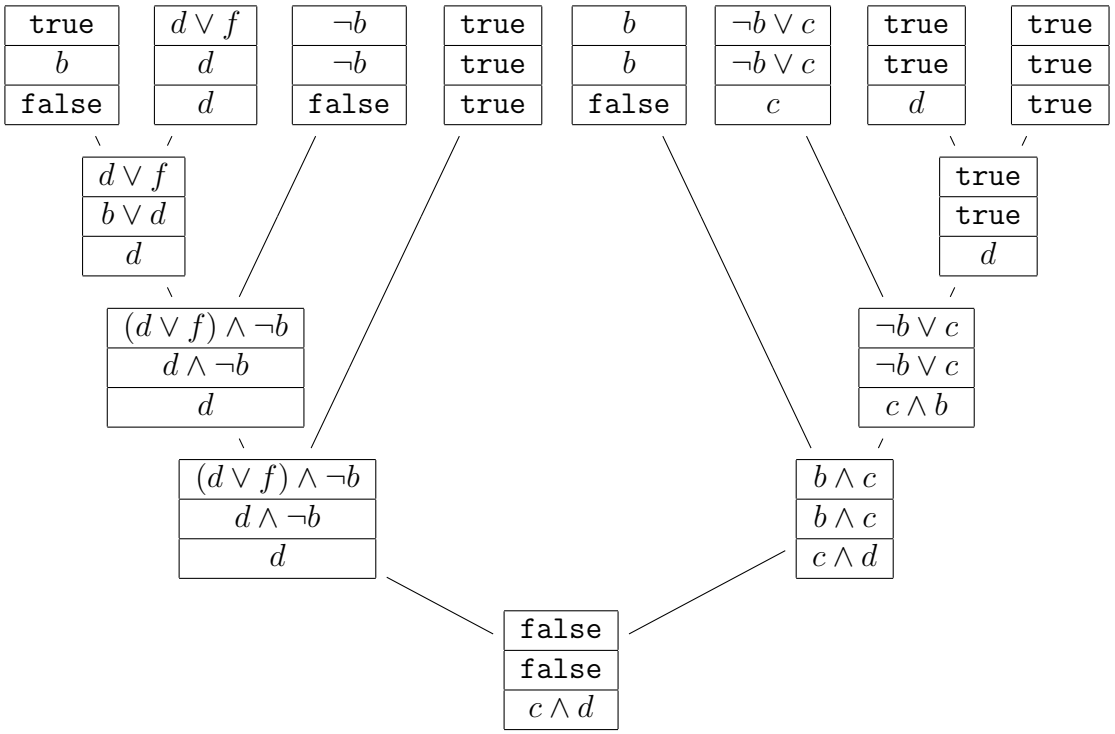


Figure 5.2.: Derived interpolants for the proof of figure 5.1

- No local variables exist, as all variables appear in A_1 as well as in B_1
- For the leaf nodes in A_1: According to definition 5.2.1, $i(c) = g(c)$ in case the node is a leaf and c is a clause from A. As there are no local variables, $g(c) = c$ for all clauses from A.
- For the leaf nodes in B_1: According to definition 5.2.1, $i(c) =$ `true` if the node is a leaf and c is a clauses from B.
- As there are no local variables, all other intermediate interpolants are obtained by conjunction of the preceeding interpolants.

□

5.3. Expressiveness of Interpolants for Abstraction Refinement

Remember the intended use of interpolants within the context of this thesis, as explained in section 1.3: Interpolants come into play if the set of formulae representing the abstract automaton and the property have been proven to be satisfiable (a counterexample is found), but the set of formulae containing the concrete formula representation, the property and the concretized trace together are not satisfiable (a spurious counterexample is found). In this case, the abstraction obviously is to coarse and has to be refined.

This section will cover the question how the sequence of interpolants can be used to identify the cause of unsatisfiability, and what additional information the sequence reveals, compared to a single interpolant. The expressiveness of the sequence of interpolants thereby strongly depends on the sequential order of the underlying formulae. Section 5.3.1 will present a sequential order of the formulae such that the interpolants reveal a maximum of information, while section 5.3.2 describes the information obtained from the interpolants.

5.3.1. Sequential Formula Order

As mentioned in the previous sections, an interpolant captures the facts the prover derived about the prefix, and which are relevant for the inconsistency of prefix and suffix. But interpolants are not unique, therefor the question arises which "kind" of interpolants will be the best for usage in model checking, that means which interpolants reveal a maximum of information?

Consider, for example, the three interpolants `true`, `false` and L, with L being a literal. With regard to section 5.1, they reveal the following information:

- `true`: Nothing can be said about the prefix, as (5.1) is always fulfilled for the interpolant `true`. From (5.2) follows: the suffix is inconsistent. No further information is revealed about the cause why prefix and suffix (together) are inconsistent.

- `false`: The prefix is inconsistent due to (5.1), but nothing is known about the suffix, as (5.2) is always fulfilled for the interpolant `false`. Again, no further information about the cause of the unsatisfiability is available.

- L: Neither prefix nor suffix are inconsistent on their own ((5.1) and (5.2)), but the pair is. The cause of the unsatisfiability is the literal L: All valuations satisfying the prefix also satisfy L (according to (5.1)), and all valuations satisfying the suffix also satisfy $\neg L$ (according to (5.2)).

With regard to weak and strong interpolants, `true` is the weakest interpolant out of the three, while `false` is the strongest (see also definition 5.1.3). In general, if `false` is ignored for the moment, strong interpolants reveal more information, as the following example shows.

Example 5.3.1 (Weak and Strong Interpolants)
Let $(\{(a \vee c), a\}, \{\neg c, \neg a\})$ be a pair of inconsistent clause sets. Possible interpolants for this pair are a and $(a \vee c)$, where a is stronger than $(a \vee c)$.

From interpolant $(a \vee c)$, it is possible to conclude that either a or c (or both) are the cause for the unsatisfiability, but interpolant a clearly states that variable a is the reason why prefix and suffix are unsatisfiable. □

This small example might help in understanding why strong interpolants reveal more information than weak interpolants do. Roughly speaking, this is because strong interpolants focus on the relevant facts which are needed to prove unsatisfiability. Although the strongest possible interpolant (`false`) does not reveal any information about the cause of unsatisfiability (as explained above), this interpolant nevertheless is helpful in the context of timed automata. This will be explained in section 5.3.2.

According to definition 3.3.11 on page 47, the k-unrolling of an automaton $\mathcal{A}$ is represented by one formula $\varphi(\mathcal{A})_k$ only (which is the conjunction of all formulae used to represent the single components). As the model checking result for this single formula (either "satisfiable" or "unsatisfiable") does not help in finding the cause of inconsistence, the formula is represented by the set of its conjunctive elements (see remark 2.1.4), that means

$$\varphi(\mathcal{A})_k = \{\varphi_{gc}(\mathcal{A})_0, \ldots, \varphi_{gc}(\mathcal{A})_k, \varphi_E(\mathcal{A})_0, \ldots, \varphi_E(\mathcal{A})_{k-1}, \varphi_i(\mathcal{A})\}.$$

In this way, interpolants can be derived for pairs of formulae (see remark 5.1.2). Note that each formula in the set contains variables of at most two consecutive unrolling depths. Although $\varphi_i(\mathcal{A})$ still is a conjunction and therefor could be divided in conjunctive elements, it will be handled as one formula. This is simply because $\varphi_i(\mathcal{A})$ does not contain free variables, and therefor will not contribute to cause of unsatisfiability.

Keeping all this in mind, the sequential order of formulae which will be given to the SAT solver can be defined.

Definition 5.3.2 (Sequential Order of Formulae)
Let $\{\varphi_{gc}(\mathcal{A})_0, \ldots, \varphi_{gc}(\mathcal{A})_k, \varphi_E(\mathcal{A})_0, \ldots, \varphi_E(\mathcal{A})_{k-1}, \varphi_i(\mathcal{A})\}$ be the set of formulae con-

taining all conjunctive elements of the k-unrolling of some automaton $\mathcal{A}$. The *sequential order* $\psi_{\mathcal{A},k}$of this set is defined as the sequence

$$\psi_{\mathcal{A},k}=(\varphi_i(\mathcal{A}), \varphi_{gc}(\mathcal{A})_0, \varphi_E(\mathcal{A})_0, \varphi_{gc}(\mathcal{A})_1, \varphi_E(\mathcal{A})_1, \ldots, \varphi_{gc}(\mathcal{A})_{k-1}, \varphi_E(\mathcal{A})_{k-1}, \varphi_{gc}(\mathcal{A})_k)$$

□

The sequence thus starts with all formulae containing only variables of unrolling depth 0, followed by the formulae with variables of unrolling depth 0 and 1, followed by those formulae containing variables of unrolling depth 1 and so on. Note that consecutive sequential orders (of the same automaton) have identical prefixes, that means the first $2 * k' + 2$ formulae of $\psi_{\mathcal{A},k}$ equal $\psi_{\mathcal{A},k'}$. This follows directly from definition 5.3.2.

Although the concretization of a counterexample will be dealt with in detail in section 6.2, it shall be mentioned here that all formulae being added to the sequence during the process of concretization preserve the sequential order (with regard to the unrolling depth).

The sequence of formulae obtained from definition 5.3.2 guarantees to "produce" quite strong interpolants. This is simply due to the fact that the interpolant contains only common variables of prefix and suffix, so the number of variables the interpolant may range over is vastly restricted.

5.3.2. Interpolant Sequences

In the preceding sections, it has been shown that strong interpolants—compared to weak interpolants for the same pair of formulae—reveal a maximum of information, and the expressiveness of single interpolants has been discussed. Now the question remains: What information exactly do sequences of interpolants contain?

For explanatory purposes of the following, let $\psi_{\mathcal{A},k} = (F_1, \ldots, F_n)$ be the sequence obtained from definition 5.3.2, let (F_i, F_{i+1}) be an abbreviation for the pair $(\{F_1, \ldots, F_i\}, \{F_{i+1}, \ldots, F_n\})$, $i \in \{1, \ldots, n\}$, and let G_i be the interpolant derived for the pair (F_i, F_{i+1}). Note that by construction, interpolant G_i will contain variables of one unrolling depth only (as interpolants refer only to common variables, see (5.3)).

In (McMillan, 2005a), the author shows for a sequence of formulae $(F_1, \ldots, F_n)$ and a sequence of interpolants $(G_1, \ldots, G_{n-1})$ such that G_i is an interpolant for the pair (F_i, F_{i+1}) and all interpolants are derived from the same refutation proof, then "G_i is sufficient to prove the interpolant for (F_{i+1}, F_{i+2})", that means

$$\models (G_i \wedge F_{i+1}) \rightarrow G_{i+1}. \tag{5.4}$$

In case G_i and G_{i+1} are seen as single interpolants, they contain information about the cause of unsatisfiability of the pair (F_i, F_{i+1}) and (F_{i+1}, F_{i+2}), respectively, as has been explained above. (5.4) expresses the fact that for a sequence of interpolants, these causes are not independent but related to each other: The cause of unsatisfiability for the pair (F_i, F_{i+1}) (which is G_i) *in addition with* the formula F_{i+1} is the cause of unsatisfiability of the pair (F_{i+1}, F_{i+2}) (which is G_{i+1}).

As mentioned above, the interpolant `false` for a pair of formulae does not express anything apart from the fact that the prefix is unsatisfiable (see (5.1)). However, this fact can be efficiently used here, as the prefixes of consecutive sequential orders (as defined in definition 5.3.2) are equal, as explained above.

Suppose the interpolant for some pair (F_i, F_{i+1}) is `false`. By definition, the prefix of the interpolant—which is given by the sequence $(F_1, \ldots, F_i)$—is inconsistent. By definition, this sequence (or the sequence $(F_1, \ldots, F_{i-1}$ in case F_i is a formula representing the transition relation) is equal to $\psi_{\mathcal{A},k'}$ for some $k' \in \mathbb{N}$, and thus $\psi_{\mathcal{A},k'}$ is unsatisfiable.

Note that for $G_i =$ `false` being the first interpolant `false`, the interpolants G_j, $j \geq i$ are not necessarily `false`, too. Consider example 5.2.2, where the interpolant $c \wedge d$ follows the two interpolants `false`. However, as the prefix already is inconsistent, all interpolants following the first interpolant `false` may be ignored, as they do not reveal any new information. Furthermore, in most SAT solvers (including FOCI), those interpolants are automatically set to `false`.

In case of unsatisfiability in the concrete case (that means a spurious counterexample has been found), the question arises which parameter(s) from the abstraction set would have to be refined to rule out this spurious counterexample[3]? The answer to this question is quite simple: Let $(G_1, \ldots, G_{n-1})$ be the sequence of interpolants, let G_j be the last interpolant (last before the first interpolant which is `false`, or last in the sequence in case not interpolant `false` exists) containing one or more parameters from the abstraction set. These parameters caused the counterexample to be spurious: As interpolants describe the cause of unsatisfiability, and the parameters which are not contained in the abstraction set have already been considered in the abstract automaton, only one of the parameters contained in G_j as well as in the abstraction set can be "responsible" for the inconsistency. For this reason, interpolants containing only parameters which are not contained in the abstraction may be ignored.

[3]Note that definitely one of the parameters from the abstraction set must be "responsible", as all other parameters are already contained in the abstract automaton. In other words: The counterexample represents a valid trace in the abstract automaton. Thus, all parameters from the abstract automaton have been taken into account while building the counterexample. Therefor, only a parameter not considered before can be responsible for the trace to be a spurious counterexample. These parameters which have not been considered before are the parameters from the abstraction set.

6. Refinement

6.1. Overview

In case a counterexample is encountered in the abstract automaton, is has to be checked whether this counterexample is real or spurious. In the former case, the property does not hold, as a counterexample has been found, and the abstraction refinement cycle terminates. In the latter case, the abstraction has to be refined—based on the reason why the abstract counterexample has been possible—in order to prevent this erroneous behaviour in further verification steps. While the preceding chapter described the information interpolants contain, this chapter deals with the question how to use this information efficiently for refinement.

To refine an abstraction, there are basically two prevalent approaches which have been widely studied: *Counterexample guided abstraction refinement* (CEGAR) (Clarke *et al.*, 2003) and *proof-based abstraction refinement* (PBAR) (McMillan & Amla, 2003). In fact, there are some other strategies—like for example probabilistic abstraction refinement (for probabilistic systems (McIver & Morgan, 2004) or thread-modular abstraction refinement (for concurrent software, (Henzinger *et al.*, 2003))—but CEGAR and PBAR are the most important two. Both of them are based on the "refutation relevance principle" (McMillan, 2005b): "Facts used to refute a class of potential models are considered relevant."

In counterexample guided abstraction refinement, one spurious abstract counterexample (corresponding to a set of concrete counterexamples, see also section 6.2 below) is ruled out within every refinement step. Thereby, refinement consists in adding facts (constraints) to the abstraction to rule out the spurious counterexample actually found. The concretization and refinement step therefor are very easy, but the number of iterations used to return a result (that means finish the abstraction refinement cycle) tends to be very large. This is due to the fact that only small model classes (sets of spurious counterexamples) are ruled out at once. Furthermore, the counterexample is not necessarily related to the property, that means irrelevant constraint may be added to the abstraction (which slows down the verification process). However, a great advantage of is the fact that the verification can be done automatically in case the initial abstraction is given.

Proof-based abstraction refinement tries to rule out all counterexamples of a given length at once. The approach is as follows: Bounded model checking is applied to the concrete system, for some bound k. In case of unsatisfiability, there exists a resolution proof expressing the fact that there is no counterexample of length k or fewer. The constraints used in this proof are considered relevant for the property, those not used are considered irrelevant (this is just a heuristic!) and are abstracted, and unbounded model checking is applied to the abstract system. In case a counterexample of length $k' > k$ (all counterexamples of length $k'' \leq k$ already have

been ruled out) is found, the concrete system is checked again with bound k'. The main advantage of PBAR is that the number of iterations needed to proof or refute a property is quite small, especially if only few parameters are relevant. Abstraction and refinement, on the other hand, require more efforts. Furthermore, the PBAR approach includes heuristics, in that not all constraints used in the refutation proof are necessarily relevant, and not all constraint *not* used in the resolution proof are necessarily irrelevant.

The next section will cover the question how to concretize a given counterexample trace, while section 6.3 will explain possible refinement strategies for the formula representation of timed automata, together with advantages and disadvantages and some heuristics where to use them.

6.2. Concretization of Counterexample

In case a counterexample trace is encountered in the abstract automaton, it has to be checked whether this is a real or spurious counterexample. This is done by concretising the trace. In general, concretising means to try to reproduce the abstract trace in the concrete system, with appropriate extensions. What exactly "reproduce" means will become clear in the sequel.

The abstract system is represented by a formula, and a trace of the abstract system is represented by a satisfying assignment to the variables contained in the formula (see also remark 3.3.12 on page 48). The trace is thus given by a list of valuations, one valuation for every parameter of the timed automaton and every unrolling depth. By definition, all variables of the abstract system are contained in the concrete system as well, therefor, the satisfying assignment of the abstract system only is a partial assignment to the concrete system.

Concretization thus means to check whether the partial assignment can be extended to a satisfying assignment of the concrete system. As there may be several satisfying assignments for the concrete system, one abstract trace may correspond to a set of concrete traces. In other words: Concretization consists in finding a valuation for the unconstrained variables such that all original constraints (given by the formulae representing the concrete automaton) are satisfied. The simplest way to do so is to express the valuations as atomic formulae, add them to the formula representation of the concrete automaton and give it to a SAT solver.

In more detail, this is realised in the following way: Let $f_{k'} = f_{k',1} \wedge \ldots \wedge f_{k',j}$ be the conjunction of the atomic formulae $f_{k',1}, \ldots, f_{k',j}$ representing variable assignments for unrolling depth k'. Remember the order of formulae explained and defined in definition 5.3.2. To preserve the property of producing strong interpolants, the atomic formulae are not simply added to the formula representation, but they are integrated in the sequence $\psi_{\mathcal{A},k}$: $f_{k'}$ is placed in the sequence between $\varphi_E(\mathcal{A})_{k'-1}$ and $\varphi_{gc}(\mathcal{A})_{k'}$. The sequence obtained in this way preserves all properties explained in section 5.3.1.

Note that indirections are not possible at the moment, that means the concrete trace has to exactly reproduce the abstract trace, without for example adding an additional state in between two states of the abstract trace.

6.3. Refining the Abstraction

Refinement comes into play when a counterexample has been identified to be spurious, that means the abstraction obviously is too coarse. The main problem in automating the refinement step is the fact that refinement—probably apart from pure CEGAR—always is a kind of heuristic: Trying to find parameters which are essential to the system in general on the basis of one abstract counterexample (or a finite set of counterexamples). Moreover, the quality of heuristic used for refinement might depend on the property to be checked.

For the abstraction by omission, as defined in chapter 4, there are two possibilities for refinement being considered here:

1. Refine a parameter, that means remove a parameter $v \in \mathcal{AS}$ from the abstraction set and add it to the system again. In this way, the abstraction becomes finer, with the hope that v is a parameter which is essential to the property.

2. Rule out a prefix of the abstract spurious counterexample actually found by reducing the set of possible traces (version of CEGAR). In this way, a set of concrete counterexamples is ruled out.

The parameter to be refined when using the first strategy may be identified with the help of interpolants using the results from section 5.3.2. In case the second strategy shall be applied, it is possible not only to rule out a single counterexample, but by ruling out a prefix of this counterexample trace, a set of (potential) counterexamples is removed. Attention has to be paid not to rule out too much. Regarding the results of section 5.3.2, the prefix which can be ruled out is determined quite easy: The prefix ends as soon as the first interpolant `false` appears.

The main task of refinement thus is to find a heuristic using both strategies and which provides a good trade-off. If an inadequate heuristic is chosen—for example using only the second possibility—the abstraction will remain coarse for a long time: In this case, many spurious counterexamples will be possible, and thus a great number of iterations will be needed to finish the abstraction refinement cycle. If, on the other hand, mainly the first possibility is used, the abstraction will become very fine after a small number of iteration steps (or might even be expanded to the concrete system in the worst case). In this case, checking the abstract system will require a great amount of time.

Possible heuristics when to chose the first approach of refining a parameter, and when to chose the second approach of ruling out a counterexample prefix are:

1. Chose the second approach in case the abstract counterexamples has a “small” length: This will rule out a great number of concrete counterexamples. The heuristic in this case is: What exactly does small mean?

   Trying this approach in examples, is has emerged that a good heuristic is to choose prefixes of traces which have at most a third of the length of the unrolling depth.

2. Chose the second approach a fixed number of times before choosing the first one. The heuristic in this case clearly is: How often should the first approach been chosen?

In combination with the first heuristic, it has been emerged that a good heuristic for this approach is to chose a fourth of the unrolling depth as number of counterexamples to be ruled out.

3. Produce a fixed number of counterexamples, and rule them out. According to the first approach above, determine the parameter which is responsible within every single counterexample. Refine the parameter which appears most frequently. The heuristic in this case is: How much counterexamples should be produced before refining a parameter?

   A good heuristic in this case is again to chose a tenth of the unrolling depth as number of counterexamples to be ruled out.

Surely, all these heuristics depend on the unrolling depth itself, the quality may greatly vary in case the unrolling depth is extremely small (< 10) or large (> 100).

7. Implementation

Within the preceding chapters, the basic principles have been established for a complete, SAT-based abstraction refinement framework. The fundamental results have been proven, and examples showed some possible applications. This chapter will present the tool SAAtRe (sat-based verification for abstraction refinement), whose development made up a big part of this thesis and which represents an implementation of the aforementioned results.

To begin with, the next section will explain some technical details of the implementation, some external tools which have been used as well as the software design of the SAAtRe tool. In sections 7.2 to 7.5, the software architecture, that means the main packages used to implement the parts of SAAtRe , is explained in detail.

7.1. Overview

The software architecture of SAAtRe directly results from figure 7.1, which is an expansion of figure 2.5 on page 34 in that it reveals more implementation details. The difference between the two is that figure 7.1 contains three additional parsers, which are used to parse the output of FOCI and Uppaal, respectively, and in this way ease data exchange. The theoretical data flow between the components has already been described in section 2.7, and is realised in the implementation using the same concepts. As mentioned in section 1.3, every "boxed" element in figure 1.1 represents a component which is conceptually different from the other components. Of course, this is still true for the boxed elements in figure 7.1, as well for the additional parser components. For this reason, every boxed component is implemented using its own package.

Note that SAAtRe itself only contains the lower part of figure 7.1, that means the cyclic component. The upper part (consisting of TA-Parser[1] and Representer, thus representing the interface between Uppaal and SAAtRe) does not belong to the SAAtRe tool itself and has not been implemented within the context of this thesis.

The implementation uses the Java programming language (Gosling *et al.*, 1996), version 1.4.2. For the internal representation of formulae as object structures in the SAAtRe implementation, the Orbital library (Platzer, 2005) has been used, which is also based on Java. Amongst others, this library provides object-oriented representations for formulae and logic, as well as other concepts (for example the concept of substitution) needed to be able to efficiently work with these formulae.

[1] TA-Parser stands for timed automaton parser, that means a parser which is used to parse the output of Uppaal for the representer to be able to convert it to the internal format of SAAtRe .

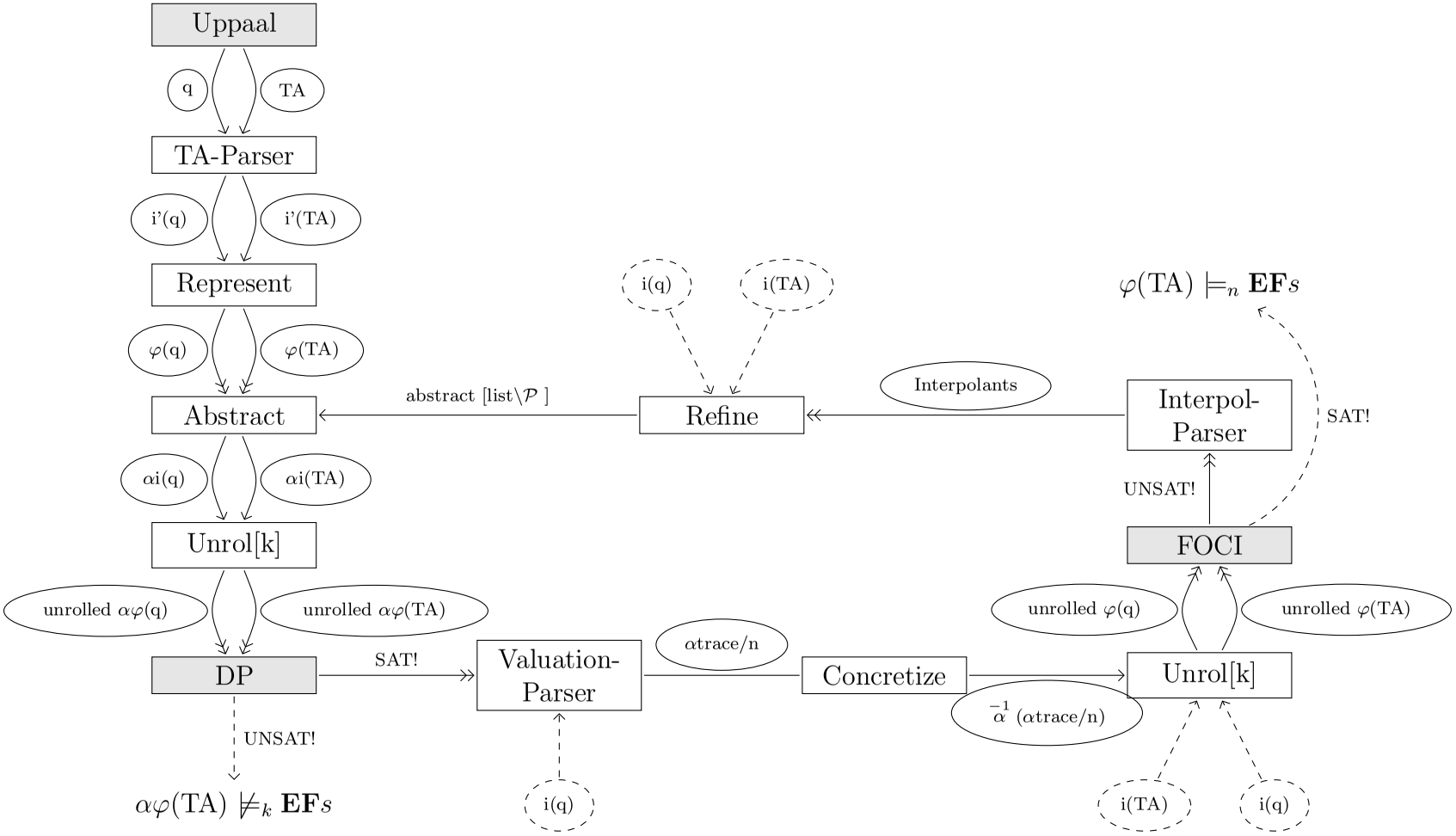


Figure 7.1.: System Architecture for SAT-based Abstraction Refinement Loop, Software Architecture

While most of the components are implemented "from scratch", the tool ANTLR (ANTLR, 2005) has been used for parser generation. Given a grammatical description (which may contain JAVA code), ANTLR (ANother Tool for Language Recognition) is able to construct a parser, a recognizer or a translator from this grammar.

For ease of compilation, the ant tool (ant, 2005) has been used which is a JAVA-based build tool. The difference to other build tools (as for example make) is that ant is not shell-based but uses an xml-based configuration file to call different targets, where each target may execute several tasks. One task thereby performs one predefined action (for example JAVADOC) on all sources specified in the build file (for example all JAVA files in the home directory).

For simplicity, the input language of SAATRE is geared to the FOCI prefix notation, as FOCI is used for checking the concrete system (see figure 7.1), that means for generating interpolants in case of unsatisfiability. A grammar describing this input language can be found in appendix A.2. The input thereby is divided in four parts, according to the conceptual seperation presented in chapterrepresentation : The 'INIT' part, with INIT being a special token, contains the representation of the inital constraints, while the 'MUTEX', 'TRANS' and 'TARGET' part contain the representation of the mutual exclusion constraint, the transition relations and the property, respectively, with MUTEX, TRANS and TARGET being special tokens. Note that with regard to further improvements of the representation, the INIT and MUTEX part are considered to be optional.

7.2. Abstracteur

The package `software.abstracteur` contains a class `Abstracteur.java` which contains a method `alpha` performing the abstraction. The abstraction implemented in the SAATRE tools works exactly like the improved abstraction by omission defined in definition 4.2.12. As the abstraction by omission, `alpha` only works for omissible parameters.

7.3. Unroller

The package `software.unroller` contains a class `Unroller.java` performing the unrolling of formulae exactly as defined in definition 3.3.11. That means: The represenation of the initial constraints is not unrolled, while—for unrolling depth k—the mutual exclusion constraint is unrolled to depth k, while the transition relation is unrolled to depth $k-1$.

7.4. Parser

As mentioned above, the parser have been implemented using the parser generator ANTLR. They are contained in the packages `software.interpol_parser` and `software.valuation_parser`.

7.5. Refiner

The Refiner contained in package `software.refiner` implements the heuristics defined in section 6.3.

8. Conclusion

Although the theory of abstraction refinement has been widely studied, and many different approaches have been investigated during the last years, there is still a lot of research to do.

Within the context of this thesis, a SAT-based approach to abstraction refinement has been motivated, explained, defined and proven.

The formula representation defined in chapter 3 is defined in a very general way, using only Boolean and rational variables. Thus, adopting this representation to other automaton and synchronisation models is quite easy. Nevertheless, the representation is sufficiently precise to allow exact representation of all components of timed automata, which has been proven.

The abstraction function defined in chapter 4 has been defined on omissible parameters and has been shown to be correct for this purpos. Abstraction from states or from whole automata has not yet been realised, but with the theoretical basis provided in chapter 4, it will be possible to define an abstraction function for parameters other than omissible ones.

In chapter 5, the expressiveness of Craig interpolants has been discussed with regard to formula sequences the interpolants are obtained from as well as with regard to abstraction refinement the interpolants are used for. The results of these chapters has been picked up in chapter 6 to define several heuristics to refine the abstraction.

Although many new approaches and ideas have been developed, defined and proven in this thesis, many possible approaches could not be realised. Among these are:

At the moment, refinement is mainly based on the first `false` interpolant, expressing that the prefix is inconsistent. It is also possible to derive information from the last `true` interpolant which expresses the fact that the suffix is inconsistent. With regard to the formula representation and the order defined in definition 5.3.2, `false` interpolants are more desirable, as they rule out the beginning of an abstract spurious counterexample trace. Nevertheless, in case `true` interpolants will appear, it should be possible in future work to derive additional information from them. Partially based on these result obtained from `true` interpolants, it should be possible to develop new (and possibly better) heuristics for refinement.

With regard to the properties SAAtRe is able to verified, future version should offer the possibility to work with full CTL, as is already realised in predicate abstraction based approaches.

As the Representer has not been implemented yet, but instead all examples have been produced manually, this is of course a task which has to be done in the future.

State variables are encoded using one Boolean variable per state, see definition 3.3.2. As it is also possible to use logarithmic encoding, this is an intersting point to start from comparing efficienty.

Concluding, abstraction refinement using SAT-based approaches is a new aspect of a widely-studied field which itself has not yet been studied in detail. For this reason, a lot of theoretical basics have to be defined when working on a framework using a SAT-based principle. But as soon as these theoretical basis will be fully defined, new results and approaches will quickly evolve.

A. Appendix

A.1. Preliminaries

A.1.1. De Morgan's Rules

De Morgan's rules describe the possibility to replace a conjunction with a discunction, and vice versa, while preserving semantics. For p, q being propositional variables, the rules are

$$\neg(p \wedge q) = \neg p \vee \neg q$$
$$\neg(p \vee q) = \neg p \wedge \neg q$$

A.1.2. Conversion to Negation Normal Form (NNF)

A formula is defined to be in negation normal form (NNF) if negations appear only in front of literals. For a formula F built with definitions 2.1.1 and 2.1.2, the NNF of F is obtained by applying function NNF to F, which is defined inductively on the structure of F. Let l be a literal, let F, F_1, F_2 be formulae.

$$\begin{aligned} NNF(l) &= l \\ NNF(\neg\neg F) &= F \\ NNF(\neg(F_1 \wedge F_2)) &= NNF(\neg F_1 \vee \neg F_2) \\ NNF(\neg(F_1 \vee F_2)) &= NNF(\neg F_1 \wedge \neg F_2) \\ NNF(F_1 \wedge F_2) &= NNF(F_1) \wedge NNF(F_2) \\ NNF(F_1 \vee F_2) &= NNF(F_1) \vee NNF(F_2) \end{aligned}$$

Thus, through repeated application of de Morgan's rules and elimination of double negation, every formula can be converted into a semantically equivalent formula in negation normal form.

A.1.3. Conversion to Conjunctive Normal Form (CNF)

A formula is defined to be in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals (see definition 2.1.3). For a formula F built with definitions 2.1.1 and 2.1.2, the CNF of F is obtained by first applying function NNF to F to obtain the negation normal form (as defined in A.1.2 above), and afterwards applying function CNF to F, which is defined inductively on the structure of F. Let l be a literal, let F, F_1, F_2 be formulae.

$$
\begin{aligned}
CNF(l) &= l \\
CNF(F \vee (F_1 \wedge F_2)) &= CNF(F \vee F_1) \wedge CNF(F \vee F_2) \\
CNF((F_1 \wedge F_2) \vee F) &= CNF(F_1 \vee F) \wedge CNF(F_2 \vee F)
\end{aligned}
$$

$CNF(F)$ preserves the semantics of F, thus, every formula can be converted to a semantically equivalent formula in conjunctive normal form.

A.1.4. Conversion to Disjunctive Normal Form (DNF)

A formula is defined to be in disjunctive normal form (DNF) if it is a disjunction of conjunctions of literals (see definition 2.1.3). For a formula F built with definitions 2.1.1 and 2.1.2, the DNF of F is obtained by first applying function NNF to F to obtain the negation normal form (as defined in A.1.2 above), and afterwards applying function DNF to F, which is defined inductively on the structure of F. Let l be a literal, let F, F_1, F_2 be formulae.

$$
\begin{aligned}
DNF(l) &= l \\
DNF(F \wedge (F_1 \vee F_2)) &= DNF(F \wedge F_1) \vee DNF(F \wedge F_2) \\
DNF((F_1 \vee F_2) \wedge F) &= DNF(F_1 \wedge F) \vee DNF(F_2 \wedge F)
\end{aligned}
$$

$DNF(F)$ preserves the semantics of F, thus, every formula can be converted to a semantically equivalent formula in disjunctive normal form.

A.2. Input Language of SAAtRe Tool

saatreinputfile → (INIT formlist)? (MUTEX formlist)?
TRANS formlist TARGET formlist
formlist → defformula ';' (defformula ';')*
defformula → relformula
| logicformula
| constformula
| atomicformula
| '(' defformula ')'
relformula → '=' defterm defterm
| '<″=' defterm defterm
| '>″=′ defterm defterm
| '>' defterm defterm
| '<' defterm defterm
logicformula → '−″ >' defformula defformula
| '<″ −″ >' defformula defformula
| '&' defformula defformula
| '‖' defformula defformula
| ¬ defformula
constformula → TRUE
| FALSE
atomicformula → VAR
| TIMEVAR
defterm → atomicterm
| arithterm
| '(' defterm ')'
arithterm → '+' '[' defterm (defterm)* ']'
| '*' NUMBER defterm
| '-' '[' defterm (defterm)* ']'
atomicterm → VAR
| TIMEVAR
| NUMBER

Index

References

2005. *Apache Ant.* release 1.6.2.
`http://ant.apache.org/`.

Aarts, Emile, & Lenstra, Jan K. (eds). 1997. *Local Search in Combinatorial Optimization.* New York, NY, USA: John Wiley & Sons, Inc.

Alur, Rajeev. 1999. Timed Automata. *Pages 8–22 of: CAV.*

Alur, Rajeev, & Dill, David L. 1994. A theory of timed automata. *Theoretical Computer Science*, **126**(2), 183–235.

ANTLR. 2005. *ANTLR parser generator.* release 2.7.5.
`http://www.antlr.org`.

Apt, Krzysztof R. 2000. *Some Remarks on Boolean Constraint Propagation.*

Artho, Cyrille, Biere, Armin, & Schuppan, Viktor. 2002 (July). Liveness checking as safety checking. *In: Formal Methods for Industrial Critical Systems (FMICS'02).*

Audemard, Gilles, Cimatti, Alessandro, Kornilowicz, A., & Sebastiani, R. 2002 (November). Bounded Model Checking for Timed Systems. *Pages 243–259 of: International Conference on Formal Techniques for Networked and Distributed Systems (FORTE).*

Biere, Armin, Cimatti, Alessandro, Clarke, Edmund M., Fujita, M., & Zhu, Y. 1999a. Symbolic model checking using SAT procedures instead of BDDs. *Pages 317–320 of: DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation.* New York, NY, USA: ACM Press.

Biere, Armin, Cimatti, Alessandro, Clarke, Edmund M., & Zhu, Yunshan. 1999b. Symbolic Model Checking without BDDs. *Pages 193–207 of: TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems.* London, UK: Springer-Verlag.

Chaki, S., Groce, Alex, & Strichman, O. 2004. Explaining abstract counterexamples. *Pages 73–82 of: SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering.* New York, NY, USA: ACM Press.

Clarke, Edmund M., & Emerson, E. Allen. 1982. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. *Pages 52–71 of: Logic of Programs, Workshop*. London, UK: Springer-Verlag.

Clarke, Edmund M., Grumberg, Orna, & Peled, Doron A. 1999. *Model checking*. Cambridge, MA, USA: MIT Press.

Clarke, Edmund M., Biere, Armin, Raimi, R., & Zhu, Y. 2001. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, **19**(1), 7–34.

Clarke, Edmund M., Grumberg, Orna, Jha, Somesh, Lu, Yuan, & Veith, Helmut. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, **50**(5), 752–794.

Cook, Stephen A. 1971. The complexity of theorem-proving procedures. *Pages 151–158 of: STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*. New York, NY, USA: ACM Press.

Courcoubetis, Costas, Vardi, Moshe Y., Wolper, Pierre, & Yannakakis, Mihalis. 1992. Memory Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, **1**, 275–288. http://www.cs.rice.edu/ vardi/papers/cav90rj.ps.gz.

Craig, William. 1957a. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *Journal of Symbolic Logic*, **22**(3), 250–268.

Craig, William. 1957b. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, **22**(3), 269–285.

Cytron, Ron, Ferrante, Jeanne, Rosen, Barry K., Wegman, Mark N., & Zadeck, F. Kenneth. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, **13**(4), 451–490.

Dantzig, George B., & Eaves, B. Curtis. 1973. Fourier-Motzkin Elimination and Its Dual. *Journal of Combinatorial Theory, Series A*, **14**(3), 288–297.

Das, S., & Dill, David L. 2001. Successive Approximation of Abstract Transition Relations. *In: Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*. June 2001, Boston, USA.

Das, S., & Dill, David L. 2002. Counter-Example Based Predicate Discovery in Predicate Abstraction. *In: Formal Methods in Computer-Aided Design*. Springer-Verlag.

Das, S., Dill, David L., & Park, S. 1999. Experience with Predicate Abstraction. *In: 11th International Conference on Computer-Aided Verification*. Springer-Verlag. Trento, Italy.

Davis, Martin, & Putnam, Hilary. 1960. A Computing Procedure for Quantification Theory. *Journal of the ACM*, **7**(3), 201–215.

Davis, Martin, Logemann, George, & Loveland, Donald. 1962. A machine program for theorem-proving. *Communications of the ACM*, **5**(7), 394–397.

Dierks, Henning. 1999. *Specification and Verification of Polling Real-Time Systems*. Ph.D. thesis, Universität Oldenburg, Germany.

FOCI. 2005. *Kenneth L. McMillan's Homepage*. `http://www-cad.eecs.berkeley.edu/~kenmcmil/`.

Fränzle, M., & Herde, C. 2005. Efficient Proof Engines for Bounded Model Checking of Hybrid Systems. *Electronic Notes in Theoretical Computer Science*, **133**, 119–137.

Gerth, R., Peled, Doron A., Vardi, Moshe Y., & Wolper, Pierre. 1996. Simple on-the-fly automatic verification of linear temporal logic. *Pages 3–18 of: Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*. London, UK, UK: Chapman & Hall, Ltd.

Gosling, James, Joy, Bill, Steele, Guy L., & Bracha, Gilad. 1996. *The Java Language Specification*. The Java Series. Massachusetts: Addison-Wesley.

Graf, Susanne, & Saïdi, Hassen. 1997. Construction of Abstract State Graphs with PVS. *Pages 72–83 of:* Grumberg, Orna (ed), *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, vol. 1254. Springer Verlag.

Groce, Alex, & Kroening, D. 2005. Making the Most of BMC Counterexamples. *Pages 67–81 of: Proceedings of Bounded Model Checking 2004*. ENTCS, vol. 119. Elsevier.

Henzinger, Thomas A., Jhala, Ranjit, Majumdar, Rupak, & Qadeer, Shaz. 2003. Thread-modular abstraction refinement. *Pages 262–274 of: Proceedings of the 15th International Conference on Computer-Aided Verification (CAV)*. Lecture Notes in Computer Science 2725, Springer-Verlag.

Henzinger, Thomas A., Jhala, Ranjit, Majumdar, Rupak, & McMillan, Kenneth L. 2004. Abstractions from proofs. *Pages 232–244 of: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press.

Kiefer, Stefan. 2005. *Abstraction Refinement for Pushdown Systems*. M.Phil. thesis, Universität Stuttgart.

Marques-Silva, João P., & Sakallah, Karem A. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, **48**(5), 506–521.

McIver, Annabelle, & Morgan, Carroll. 2004. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. SpringerVerlag.

McMillan, Kenneth L. 1993. *Symbolic Model Checking*. Ph.D. thesis, Norwell, MA, USA.

McMillan, Kenneth L. 2003 (July). Interpolation and SAT-based Model Checking. *In: Computer Aided Verification (CAV03)*.

McMillan, Kenneth L. 2004. An Interpolating Theorem Prover. *Pages 16–30 of: Tools and Algorithms for Construction and Analysis of Systems (TACAS)*.

McMillan, Kenneth L. 2005a. Applications of Craig Interpolants in Model Checking. *Pages 1–12 of: Tools and Algorithms for Construction and Analysis of Systems (TACAS)*.

McMillan, Kenneth L. 2005b (July). *Tutorial on Automated Abstraction Refinement*.

McMillan, Kenneth L., & Amla, Nina. 2003. Automatic Abstraction without Counterexamples. *Pages 2–17 of: Tools and Algorithms for Construction and Analysis of Systems (TACAS)*.

McMillan, Kenneth L., & Jhala, Ranjit. 2005. Interpolant-based Transition Relation Approximation. *In: Computer Aided Verification (CAV05)*.

Moskewicz, Matthew W., Madigan, Conor F., Zhao, Ying, Zhang, Lintao, & Malik, Sharad. 2001. Chaff: Engineering an Efficient SAT Solver. *In: Proceedings of the 38th Design Automation Conference (DAC'01)*.

Platzer, André. 2005. *Orbital library*. `http://www.functologic.com`.

Pudlák, Pavel. 1997. Lower Bounds for Resolution and Cutting Plane Proofs and Monotone Computations. *Journal of Symbolic Logic*, **62**(3), 981–998.

Robinson, J. A. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, **12**(1), 23–41.

Schöning, Uwe. 1995. *Logik für Informatiker*. Fourth edn. Spektrum Verlag.

Schöning, Uwe. 2003. *Theoretische Informatik – kurzgefasst*. First edn. Spektrum Verlag.

Sorea, Maria. 2003. *Verification of Real-Time Systems through Lazy Approximations*. Ph.D. thesis, Universität Ulm, Germany.

Thomas, Wolfgang. 1990. Automata on Infinite Objects. *Pages 133–192 of: Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*.

uppaal. 2005. *Uppaal*. release 3.4.11. `http://www.uppaal.com/`.

Wolper, Pierre. 2002. Constructing automata from temporal logic formulas: a tutorial. *Lectures on formal methods and performance analysis: first EEF/Euro summer school on trends in computer science*, 261–277.

Zhang, Hantao. 1997. SATO: An Efficient Propositional Prover. *Pages 272–275 of: CADE-14: Proceedings of the 14th International Conference on Automated Deduction*. London, UK: Springer-Verlag.

Zhang, Lintao, & Malik, Sharad. 2003 (March). Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. *In: Proceedings of Design, Automation and Test in Europe (DATE2003)*.