# Verification of Timed Automata by CEGAR-Based Algorithms

Scientific Students' Associations Report

Author:

Rebeka Farkas

Supervisors:

András Vörös
Tamás Tóth
Ákos Hajdu

2016.

# Contents

**Kivonat**  A napjainkban egyre inkább elterjedő biztonságkritikus rendszerek hibás működése súlyos károkat okozhat, emiatt kiemelkedően fontos a matematikailag precíz ellenőrzési módszerek alkalmazása a fejlesztési folyamat során. Ennek egyik eszköze a formális verifikáció, amely már a fejlesztés korai fázisaiban képes felfedezni tervezési hibákat. A biztonságkritikus rendszerek komplexitása azonban gyakran megakadályozza a sikeres ellenőrzést, ami különösen igaz az időzített rendszerekre : akár kisméretű időzített rendszereknek is hatalmas vagy akár végtelen állapottere lehet. Ezért különösen fontos a megfelelő modellezőeszköz valamint hatékony verifikációs algoritmusok kiválasztása. Az egyik legelterjedtebb formalizmus időzített rendszerek leírására az időzített automata, ami a véges automata formalizmust óraváltozókkal egészíti ki, lehetővé téve az idő múlásának reprezentálását a modellben.

Formális verifikáció során fontos kérdés az állapotelérhetőség, amely során azt vizsgáljuk, hogy egy adott hibaállapot része-e az elérhető állapottérnek. A probléma komplexitása már egyszerű (diszkrét változó nélküli) időzített automaták esetén is exponenciális, így nagyméretű modellekre ritkán megoldható. Ezen probléma leküzdésére nyújt megoldást az absztrakció módszere, amely a releváns információra koncentrálva próbál meg egyszerűsíteni a megoldandó problémán. Az absztrakció-alapú technikák esetén azonban a fő probléma a megfelelő pontosság megtalálása. Az ellenpélda vezérelt absztrakciófinomítás (counterexample-guided abstraction refinement, CEGAR) iteratív módszer, amely a rendszer komplexitásának csökkentése érdekében egy durva absztrakcióból indul ki és ezt finomítja a kellő pontosság eléréséig.

Munkám célja hatékony algoritmusok fejlesztése időzített rendszerek verifikációjára. Munkám során az időzített automatákra alkalmazott CEGAR-alapú elérhetőségi algoritmusokat vizsgálom és közös keretrendszerbe foglalom, ahol az algoritmusok komponensei egymással kombinálva új, hatékony ellenőrzési módszerekké állnak össze. Az irodalomból ismert algoritmusokat továbbfejlesztettem és hatékonyságukat mérésekkel igazoltam.

**Abstract**   Nowadays safety-critical systems are becoming increasingly popular, however, faults in their behavior can lead to serious damage. Because of this, it is extremely important using mathematically precise verification methods during their development. One of these methods is formal verification that is able to find design problems since early phases of the development. However, the complexity of safety-critical systems often prevents successful verification. This is particularly true for real-time systems: even small timed systems can have large or even infinite states pace. Because of this, selecting an appropriate modeling formalism and efficient verification algorithms is very important. One of the most common formalism for describing timed systems is the timed automaton that extends the finite automaton with clock variables to represent the elapse of time.

When applying formal verification, reachability becomes an important aspect – that is, examining whether or not the system can reach a given erroneous state. The complexity of the problem is exponential even for simple timed automata (without discrete variables), thus it can rarely be solved in case of large models. Abstraction can provide assistance by attempting to simplify the problem to solve while focusing on the relevant information. In case of abstraction-based techniques the main difficulty is finding the appropriate precision. Counterexample-guided abstraction refinement (CEGAR) is an iterative method starting from a coarse abstraction and refining it until the sufficient precision is reached.

The goal of my work is to develop efficient algorithms for verification of timed automata. In my work I examine CEGAR-based reachability algorithms applied to timed automata and I integrate them to a common framework where components of different algorithms are combined to form new and efficient verification methods. I improved known algorithms and proved their effectivity by measurements.

**TODO:** Ákos-javítások

# Chapter 1

# Introduction

**TODO:** Abstract+ kis módosítás

Chapter 2

# Background

## 2.1 Mathematical logic

Mathematical logic is useful for deciding correctness of systems. This section provides some insight about propositional logic, first order logic, and the satisfiability problem. Difference logic is also introduced.

### 2.1.1 Propositional logic

Propositional logic (or zeroth order logic) is concerned with the study of formulae of boolean variables, and deciding whether they are true or false. Propositional formulae are composed of truth symbols $\top$ (true) and $\bot$ (false), and propositional variables $p, q, \ldots$ with the use of logical connectives. A formula $\varphi$ can be an atom (a truth symbol or a variable) or can be constructed from other logical formulae with the following connectives:

- negation: $\neg\varphi$ is evaluated true iff $\varphi$ is evaluated false (formal equivalent of 'not'),

- conjunction: $\varphi_1 \wedge \varphi_2$ is evaluated true iff both $\varphi_1$ and $\varphi_2$ is evaluated true (formal equivalent of 'and'),

- disjunction: $\varphi_1 \vee \varphi_2$ is evaluated true iff at least one of $\varphi_1$ and $\varphi_2$ is evaluated true (formal equivalent of 'or'),

- implication: $\varphi_1 \rightarrow \varphi_2$ is evaluated true iff $\varphi_1$ is evaluated false or both $\varphi_1$ and $\varphi_2$ is evaluated true (formal equivalent of 'if ... then'),

- equivalence: $\varphi_1 \leftrightarrow \varphi_2$ is evaluated true iff both $\varphi_1$ and $\varphi_2$ is evaluated true or both $\varphi_1$ and $\varphi_2$ is evaluated false (formal equivalent of 'if and only if').

*Note:* disjunction, implication, and equivalence can be expressed using negation and conjunction. These operators are only defined to raise the level of descriptive power.

The (boolean) satisfiability problem (SAT, for short) can be defined as follows.

**Input** : A propositional logic formula $\varphi$.

**Output** : *Yes* if $\varphi$ is satisfibile (i.e. it is possible to ground the variables appearing in $\varphi$ to truth symbols so that $\varphi$ is evaluated true), *No* if not.

### 2.1.2   First order logic

Propositional logic is useful, however, sometimes its expressive power is not enough. First order logic is a extends propositional logic with predicates, functions and the quantifiers $\exists$ (existential quantifier) and $\forall$ (universal quantifier). While propositional logic formulae could only be evaluated to truth values, first order logic formulae can be evaluated to numbers, or any abstract concept.

The basic elements of first order logic are *terms*. Variables and constants (0-ary functions) are terms, as well as *n*-ary functions applied *n* terms. In first order logic an atom can be $\top, \bot$ or an a *n*-ary predicate applied to *n* terms. Formulae are gained by applying connectives (the same as in case of propositional logic) and quantifiers to atoms.

The satisfiability problem can be applied to first order logic formulae, but it is undecidable. However, there is a variant of the problem that is applicable, and solvable for most practical problems. The key idea is to formalize structures.

> **Definition 2.1**   A *first order theory* $\mathcal{T}$ is a pair $(\Sigma, \mathcal{A})$ where
>
> - $\Sigma$ is the *signature*, i.e. the set of constant, function and predicate symbols and
>
> - $\mathcal{A}$ is the set of *axioms* where an axiom is a first order logic formula that has no quantifiers in it, and $\Sigma$ contains all constants, functions and predicates appearing in it.

The *Satisfiable Modulo Theories* (SMT) for short can be defined as follows.

**Input** : A theory $\mathcal{T} = (\Sigma, \mathcal{A})$, and $\Sigma$-formula $\varphi$.

**Output** : *Yes* if $\varphi$ is satisfiable in $\mathcal{T}$, *No* if not.

In many practical theories, SMT becomes solvable.

### 2.1.3   Difference logic

An atom in (integer) difference logic is a logical expression of the form $x - y < n$ or $x - y \leq n$ where $x$ and $y$ are variables defined over $\mathbb{Z}$ and $n$ is a constant. A difference logic formula $\varphi$ is a conjunction of one ore more atoms. In case of difference logic, SAT is not only decidable, but polynomial.

Since framework presented in this paper relies on a SAT (SMT) solver (as it is explained in chapter 4), this paper does not address the algorithms for deciding satisfiability (if it is decidable). For more information on satisfiability, the reader is referred to [2].

## 2.2   Formal verification

Formal verification is the act of proving the correctness or incorrectness of a system in a mathematically precise way. Model checking is an automatic formal verification technique, where the system has to be represented by a formal model, and the requirements of the system has to be a property of the system formally defined as logical formulae. Verification can be performed by proving that the system (represented by the formal model) satisfies that property.

### 2.2.1   Modeling formalisms

It is important to find the appropriate representation of the model i.e. the appropriate formalism in order to be able to model and check the property. This paper focuses on *behavior properties* examined on *behavioral models*.

#### Finite automata

The finite automaton (plural: automata) is one of the most common formalisms for modeling behavior. This way the system is described by a finite set of possible states, and a set of steps defining the system's state changes.

> **Definition 2.2**   Formally a *finite automaton*, or state machine $\mathcal{A}$ is a tuple $\langle S, s_0, T \rangle$ where
>
> - $S$ is a finite set of states,
>
> - $s_0 \in S$ is the initial state and
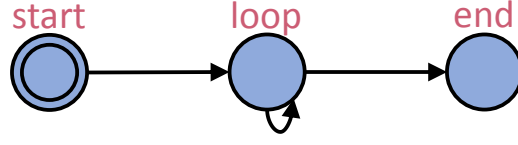>
> - $T \subseteq (S \times S)$ is a set of transitions.

**Figure 2.1**    Example of a finite automaton

Structurally $\mathcal{A}$ can be represented as a directed graph $G_A$ where $V(G_A) = S$ and $E(G_A) = T$. The system's operation is described as follows.

Initially, the system is in $s_0$. The system can change its state to some other state $s_1$ iff $(s_0, s_1) \in T$. from $s_1$ it can change its state to $s_2$ iff $(s_1, s_2) \in T$, and so on.

**Example 2.1**    Let $\mathcal{A}$ be a finite automaton with states *start*, *loop*, and *end*, and transitions *start* → *loop*, *loop* → *loop*, and *loop* → *end*, as depicted on Figure 2.1. The initial state is *start*, denoted by double outline.

Operation starts form the initial state, *start*. The system can step to *loop* where it can step to *end* or *loop* (itself). Since it is always a possibility to stay in the current state, loop transitions don't have importance in finite automata.

This formalism is easy to use and verify, but its expressive power is not necessary: there are many types of behaviours that can't be modelled this way.

### Finite automaton extended with variables

Many extensions of the finite automaton are known with various levels of expressive power. The following extension lifts the level of expressive power to that of Turing-machines: extending the automaton with variables.

**Definition 2.3**    A finite automaton extended by variables could be (briefly) defined as a tuple $\langle L, l_0, v_0, E, I \rangle$ operating on a set of variables $\mathcal{V}$ where

- $L$ is the set of control locations,

- $l_0 \in L$ is the initial location,

- $v_0$ is a function assigning an initial value to each variable $x \in \mathcal{V}$,

- $E \subseteq L \times \mathcal{B} \times \mathcal{U} \times L$ is the set of edges (where $\mathcal{B}$ can be briefly described as the set of bool valued first order logic formulae constructed from the variables of $\mathcal{V}$, and $\mathcal{U}$ can briefly defined as the set of unary functions operating on subsets of $\mathcal{V}$ assigning new values to variables) and

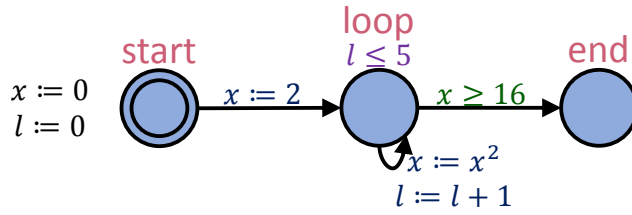- $I : L \rightarrow \mathcal{B}$ assigns invariants to locations.

**Figure 2.2** Extended finite automaton

Instead of states, the graph's nodes are called control locations, since the system's state also depends on the values of the variables. For similar reasons, the edges of the graph are now called edges in the formalism as well, and they are more expressive: an edge $e = (l, g, a, l')$ represents a transition form $l$ to $l'$, with a guard $g$ and an assignment function $a$. A guard is a condition that has to be satisfied in order for the transition to be enabled. The function $a$ describes how the value of some variables is changed during the transition. Locations can have invariants, that are conditions that have to be satisfied while the system stays in that location.

The system's operation starts from the control location $l_0$, and the variables are initialized as $v_0$ defines. A system can transition from $l$ to some $l'$ if there exist an edge $e = (l, g, a, l')$ where $g$ is satisfied by the variables current values, and $I(l')$ is satisfied by the values $a$ assigns to the variables (or their current value if $a$ is undefined on them).

**Example 2.2** Let $\mathcal{A}$ be an extended version of the automaton on figure 2.1 as depicted in figure 2.2. The introduced variables are $x$ and $l$. The variable $l$ is used as a loop counter. The edge *start* $\rightarrow$ *loop* assigns 2 to $x$, *loop* $\rightarrow$ *loop* squares $x$ and increases the value of the loop counter, *loop* $\rightarrow$ *end* doesn't affect the values of the variables, but it is only enables when $x$ is a least 16. Location *loop* has an invariant ensuring that the loop edge is taken at most 5 times.

Operation starts in state *start*, $x = 0, l = 0$. When the system changes its state it can only take edge *start* $\rightarrow$ *loop*, resulting in state *loop* $x = 2, l = 0$. Edge *loop* $\rightarrow$ *end* is not enabled, but the system can take the loop edge resulting in state *loop* $x = 4, l = 1$. The edge to *end* is still not enabled, but after taking the loop edge again (*loop* $x = 16, l = 2$) it becomes enabled. The system may keep taking the loop edge up to three times (after which the invariant prohibits it, because it would increase $l$), or transition to *end*.

## 2.2.2  Reachability

During formal verification, one of the most important questions is reachability: deciding whether a system can step into a given state. In many cases, the state in question represents an erroneous state and the desired outcome of model checking is that it is unreachable.

The problem can be defined as follows.

**Input** : A system $S$ and a state $s_{err}$.

**Output** : *Yes* if it is possible for $S$ to operate in a way that it eventually steps in $s_{err}$, *No* if not.

When the answer is *Yes*, it is useful to provide a *counterexample*: an execution trace $\sigma = s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_{err}$ setting the system's state to $s_{err}$ where $s_i$-s are states and $s_i \rightarrow s_{i+1}$ notations represent possible steps of the system from $s_i$ to $s_{i+1}$.

Each formalisms have their own interpretation of the problem - regarding how erroneous states and execution traces are described. For example, in case of finite automata, the problem can be interpreted as follows.

**Input** : A finite automaton $\mathcal{A} =, s_0, T\rangle$ and a state $s_{err} \in S$.

**Output** : A sequence of states and transitions $\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} s_{n+1} = s_{err}$ $(s_i \in S, t_i = (s_i, s_{i+1}) \in T$ for all $0 \geq i \geq n)$ if the $\mathcal{A}$ can reach $s_{err}$ , *No* if not.

This problem can be solved by any pathfinding algorithm executed on $G_{\mathcal{A}}$.

**Example 2.3**  Let the input be the previous automaton on figure 2.1 and it's state *end*. A pathfinding algorithm finds a path, e.g. *start* $\rightarrow$ *loop* $\rightarrow$ *end*, proving the state is reachable.

In case of extended finite automata the problem can be interpreted as follows.

**Input** : A finite automaton $\mathcal{A} = \langle L, l_0, v_0, E, I \rangle$ and a control location $l_{err} \in L$.

**Output** : A sequence of locations and enabled transitions $\sigma = l_0 \xrightarrow{e_0} l_1 \xrightarrow{e_1} \cdots \xrightarrow{e_n} l_{n+1} = l_{err}$ if the $S$ can reach $s_{err}$, *No* if not.

Note, that in this case error states are defined solely by the location, however, it is easy to reduce a problem where the values of the variables are also constrained into this form.
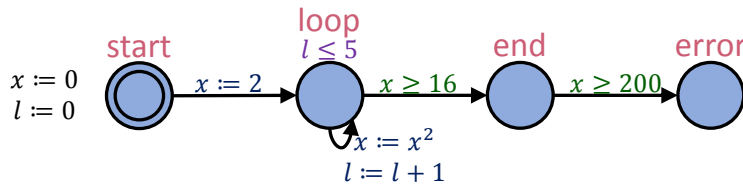
**Figure 2.3**    Automaton for checking reachability

**Example 2.4**    Consider the previous system $\mathcal{A}$ depicted on figure 2.2 with the erroneous states defined as the set of states where the current location is *end* and $x \geq 200$.

A new automaton can be constructed by extending $\mathcal{A}$'s set of location with a new location *error* that is only reachable if $\mathcal{A}$ can reach location *end* with $x \geq 200$. The new automaton is depicted on figure 2.3.

The described problem is unsolvable as well as the reachability problem for all other Turing-complete formalisms. This is one of the reasons why the modeling formalism has to be chosen carefully. A 'simple' formalism may not have enough expressive power to precisely model the system, while verification of more complex formalisms may be ineffective or even impossible.

### Statespace exploration

Even if reachability is undecidable (or just ineffective), there are many methods and approaches on how to gain useful information on the problem. The most obvious approach is *statespace exploration*.

**Definition 2.4**    The *statespace* of a system is the set of states that are reachable from the initial state by a sequence of enabled transitions.

The idea of statespace exploration is to systematically list all possible states in the statespace. If the erroneous state is found the system is proven to be incorrect. Otherwise if all possible states are listed (in case of a finite statespace) and no erroneous state is found, the system is proven to be correct. (In case of an infinite statespace this process may never terminate.)

One of the simplest ways for exploring the statespace is by constructing a search-tree. The root of the tree is the initial state $s_0$ of the system. The statespace is explored by iteratively choosing a leaf with a state $s$ of the tree, and introducing a new edge for all possible enabled transitions $(s, s')$ pointing to a new node with a state $s'$. This way, all possible execution traces are explored, but states may appear more than once if there
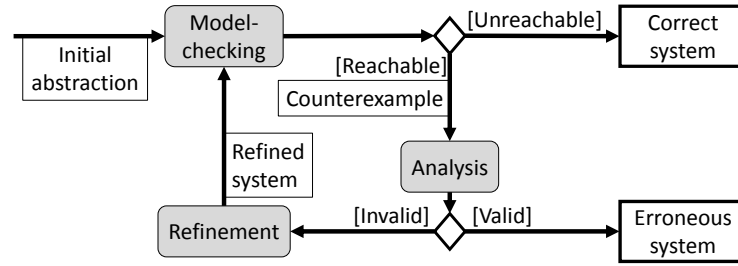
**Figure 2.4**   Counterexample-guided abstraction refinement

are more execution traces to reach them.

It is also possible to construct a state graph, where the statespace is explored similarly, except one state can only appear on one node. This helps reducing the size of the graph, however (in case of an infinite statespace) it still might be infinite. Infinite statspaces can never be completely explored this way (by explicit exploration), however, sometimes, when small counterexamples are expected, it is not necessary.

Consider the *bounded reachability problem*.

**Input** : A system $S$, a state $s_{err}$, and a bound $k$.

**Output** : A counterexample, if $S$ can reach $s_{err}$ in at most $k$ steps, *No* if not.

This problem is decidable, even for finite automata with variables. Even so, explicit statespace exploration can not be considered an effective method.

### 2.2.3   CEGAR

One of the possible approaches to perform more effective model checking is abstraction. A less detailed system model is constructed that hides unimportant parts of the behavior providing a model of complex statespace overapproximating the original one. The idea of counterexample guided abstraction refinement (CEGAR) [3] is to apply model checking to this simpler model, and then examine the results on the original one. The idea is illustrated on Figure 2.4.

First, an abstract system is constructed. The key property of abstraction is that the state space of the abstract system overapproximates that of the original one, but it is less complex and thus model checking can be performed more effectively on the abstract system.

Model checking is performed on the abstract model. If the target state is unreachable in the abstract model, it is unreachable in the original model as well. Otherwise the model checker produces a counterexample – an execution trace demonstrating how the system can reach the target state.

Overapproximation brings such behaviours to the system that are not feasible in the original one. Because of this, the counterexample may not be a valid trace in the real system, so it has to be investigated. If it turns out to be a feasible counterexample, the target state is reachable. Otherwise the abstract system has to be refined – hidden details of the original system have to be reintroduced to the model. The goal of the refinement is to modify the abstract system so that it remains an abstraction of the original one, but the spurious counterexample is eliminated. Model checking is performed on the refined system, and the CEGAR loop starts over.

The algorithm terminates when no more counterexamples are found or when a feasible trace is presented leading to the erroneous state.

### Example

There are many ways CEGAR can be implemented. An approach is presented here that can be applied to a wide range of formalisms.

When constructing a CEGAR-based algorithm the most important decision to make is the way abstraction is applied to the system. It determines the class of algorithms that can be used for model checking, the nature of the counterexample, and how it can be analyzed, and the possibilities for refinement.

Abstraction can be applied to many aspects of a model. An abstraction method commonly used for formalisms operating on variables is to reduce the number of variables in the model, by simply ignoring some of them. The initial abstraction of the model can be the same model without any variables.

> **Example 2.5**  Consider again automaton $\mathcal{A}$ depicted on figure 2.2. Eliminating all variables results in the previous finite automaton depicted on figure 2.1.

Model checking can be performed by statespace exploration for example, however, there are countless other ways.

> **Example 2.6**  In the first iteration model the statespace exploration of $\mathcal{A}$ will result in $G(\mathcal{A})$, where the counterexample found can be e.g. $\sigma = start \rightarrow loop \rightarrow end$.

A common way of checking whether a counterexample is feasible is by transforming it into a first order logic formula, and handing it to a solver. If it is satisfiable, the error location is reachable, and the system is incorrect. Consider for example the case of the extended finite automaton. The counterexample is a trace $\sigma = l_0 \xrightarrow{e_0} l_1 \xrightarrow{e_1} \cdots \xrightarrow{e_n} l_{n+1} = l_{err}$.

Variables have to be defined: for each $x \in \mathcal{V}$ variables $x_0, x_1, \ldots, x_n$ and $x_{err}$ are defined - one for each state on the counterexample. Afer that a set of constraints is constructed to define the automaton's behaviour.

First, initial conditions are defined. In case of extended finite automata, the initial constraints are the variables' initial values. For each $x \in \mathcal{V}$ $x_0 = v_0(x)$ is added to the set of constraints. After that, constraints are added step by step.

It has to be checked if the guard is satisfied. For each $e_i = (l_i, g_i, a_i, l_{i+1})$ $0 \leq i \leq n$ a variation of $g_i$ is added to the set of constraints: all $x \in \mathcal{V}$ appearing in $g_i$ is replaced by $x_i$. Afer that, the assignment function is considered: for each $x \in \mathcal{V}$ $x_{i+1} = a_i(x)$ is added to the set of constraint if $a_i(x)$ exists, and $x_{i+1} = x_i$ is added otherwise. Invariants can be turned into constraints the same way as guards: all $x \in \mathcal{V}$ appearing in $I(l_i)$ is replaced by $x_i$.

> **Example 2.7**  The translation of the counterexample $\sigma = start \rightarrow loop \rightarrow end$ to a logic formula will contain variables $x_0, x_1, x_2, l_0, l_1$ and $l_2$. The initial constraints are $x_0 = 0, l_0 = 0$. Edge $start \rightarrow loop$ assigns 2 to $x$, but leaves $l$ unchanged, yielding the constraints $x_1 = 2, l_1 = l_0$. Location $loop$'s invariant $l \leq 5$ can be transformed to constraint $l_1 \leq 5$. The guard of $loop \rightarrow end$ yields the constraint $x_1 \geq 16$.
>
> The formula handed to the solver is $\varphi = x_0 = 0 \bigwedge l_0 = 0 \bigwedge x_1 = 2 \bigwedge l_1 = l_0 \bigwedge l_1 \leq 5 \bigwedge x_1 \geq 16$.

The conjunction of the resulting set of constraints is handed to a solver. If it is satisfiable, the counterexample is feasible, and the erroneous state is reachable. Otherwise it is a spurious counterexample and the abstract system has to be refined. Solvers have useful features for this as well.

> **Example 2.8**  The formula $\varphi = x_0 = 0 \bigwedge l_0 = 0 \bigwedge x_1 = 2 \bigwedge l_1 = l_0 \bigwedge l_1 \leq 5 \bigwedge x_1 \geq 16$ is obviously unsatisfiable, because $x_1 = 2 \wedge x_1 \geq 16$ is unsatisfiable in itself. If the automatons current abstraction is extended with the variable $x$ with all of its assignments, and all constraints in guards and invariants that $x$ appears but $l$ doesn't, this counterexample won't be found anymore.

## 2.3   Timed automata

The timed automaton is a common formalism for modeling timed systems. It is an extension of the finite automaton with clock variables. In this section clock variables and timed automata are introduced, an algorithm is described (and the implementation briefly explained) for deciding reachability, and information is provided on the complexity of the problem.

### 2.3.1  Basic Definitions

In order to properly define timed automata, first the idea of *clock variables* must be explained. In case of systems with discrete variables, the values of the variables always remain the same between two modifications. However, this is not the case for clock variables (clocks, for short).

> **Definition 2.5**  *Clock variables* are a special type of variables, whose value is constantly and steadily increasing.

When a system stays in one state, the value of clocks are increasing. Naturally, their values can be modified, but the only allowed operation on clock variables is *reset*. Resetting a clock means assigning its value to 0. It's an instantaneous operation, after which the value of the clock will continue to increase.

Hereinafter follows some basic definitions that are closely related to clock variables and timed automata.

> **Definition 2.6**  A *valuation* $v(\mathcal{C})$ assigns a non-negative real value to each clock variable $c \in \mathcal{C}$, where $\mathcal{C}$ denotes the set of clock variables.

In other words a valuation defines the values of the clocks at a given moment of time. The term *valuation* can also be used for discrete variables.

> **Definition 2.7**  A *clock constraint* is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$ (*difference constraint*), where $x, y \in \mathcal{C}$ are clock variables, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. $\mathcal{B}(\mathcal{C})$ represents the set of clock constraints.

In other words a clock constraint defines upper and lower bounds on the values of clocks (or differences of clocks, in case of difference constraints). Bounds are always integer numbers.

A *timed automaton* extends a finite automaton with clock variables. It can be defined as follows.

> **Definition 2.8**  A *timed automaton* $\mathcal{A}$ is a tuple $\langle L, l_0, E, I \rangle$ where
>
> - $L$ is the set of locations,
>
> - $l_0 \in L$ is the initial location,
>
> - $E \subseteq L \times \mathcal{B}(\mathcal{C}) \times 2^{\mathcal{C}} \times L$ is the set of edges and
>
> - $I : L \to \mathcal{B}(\mathcal{C})$ assigns invariants to locations. [1]
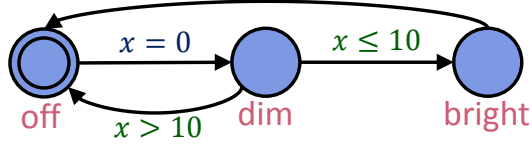
**Figure 2.5**   Timed automaton model of a lamp

The automaton's edges, that are defined by the source location, the guard (represented by a clock constraint), the set of clocks to reset (the timed equivalent of the assignment function), and the target location.

**Example 2.9**   The timed automaton depicted on figure 2.5 models a lamp with three levels of intensity: off (no light), dim, and bright. When it is turned off, one push of the button turns the dim light on, two quick (in 10 time units) pushes of the button turn the bright light on. When it is turned on, it can be switched off with a push of the button.

**Definition 2.9**   A *state* of $\mathcal{A}$ is a pair $\langle l, v \rangle$ where $l \in L$ is a location and $v : \mathcal{C} \to \mathbb{R}$ is the current valuation satisfying $I(l)$.

In the initial state $\langle l_0, v_0 \rangle$ $v_0$ assigns 0 to each clock variable.

Two kinds of operations are defined.

**Definition 2.10**   The state $\langle l, v \rangle$ has a *discrete transition* to $\langle l', v' \rangle$ if there is an edge $e(l, g, r, l') \in E$ in the automaton such that

- $v$ satisfies $g$,

- $v'$ assigns 0 to any $c \in r$ and assigns $v(c)$ to any $c \notin r$, and

- $v'$ satisfies $I(l')$.

**Definition 2.11**   The state $\langle l, v \rangle$ has a *time transition* (or delay, for short) to $\langle l, v' \rangle$ if

- $v'$ assigns $v(c) + d$ for some non-negative $d$ to each $c \in \mathcal{C}$ and

- $v'$ satisfies $I(l)$.

**Example 2.10**   The automaton of the previously mentioned lamp operates on one clock variable, $x$. Initially $x = 0$ but as long as the system stays in location *off*, its value increases and it can reach any non-negative value. Once it steps to *dim x* is reset, and its value increases from 0 again. The next discrete transition is decided by the amount of time the system spent in location *dim*. If it is at most ten time units, the system steps into location *bright* and continues to increase from its last value in *dim*, that can be any value between 0 and 10. Otherwise, the system steps back to *off* and the value of $x$ continues to increase from its last value in *dim*, that is more than 10.

There are many extensions of the timed automata formalism. Most of them – such as network automata, synchronization, and urgent locations – can be easily transformed into conventional timed automata, but this is not always the case. The idea to allow discrete variables as well as clock variables arises simply, but the same way as in case of finite automaton, growing expressive power yields less efficient model checking.

### 2.3.2   Timed automaton reachability

In case of timed automata the reachability problem can be defined as follows.

**Input**  : An automaton $\mathcal{A} = \langle L, l_0, E, I \rangle$, and a location $l_{err} \in L$.

**Output**  : An execution trace $\sigma = l_0 \xrightarrow{t_0} l_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} l_{err}$ from $l_0$ to $l_{err}$ or *NO*, if it is unreachable.

The problem is decidable. One of the most effective algorithm for deciding reachability is the algorithm used by *Uppaal*[1], a model checker for timed automata. The core of the algorithm is published in [1].

### Algorithm

Before presenting the algorithm, some basic definitions have to be provided. First, zones are introduced as an abstract domain for clock valuations.

**Definition 2.12**   A *zone z* is a set of nonnegative clock valuations satisfying a clock constraint.

**Definition 2.13**   A *zone graph* is a finite graph consisting of $\langle l, z \rangle$ pairs as nodes, where $l \in L$ refers to some location of a timed automaton and $z$ is a zone. Edges of the zone graph represent transitions.
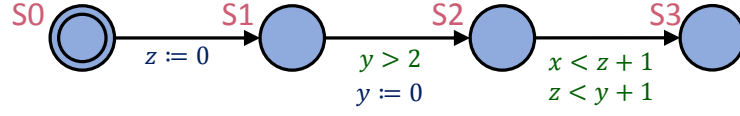
---

[1]http://www.uppaal.org/

**Figure 2.6**   Timed automaton

A node $\langle l, z \rangle$ of a zone graph represents all states $\langle l, v \rangle$ where $v \in z$. Since edges of the zone graph denote transitions, a zone graph can be considered as an (exact) abstraction of the statespace. The main idea of the algorithm is to explore the zone graph of the timed automaton, and if a node $\langle l_{err}, z \rangle$ exists in the graph for some $z \neq \emptyset$, $l_{err}$ is reachable, and the execution trace can be provided by some path-finding algorithm.

The construction of the graph starts with the initial node $\langle l_0, z_0 \rangle$, where $l_0$ is the initial location and $z_0$ contains the valuations reachable in the initial location by time transitions. Next, for each outgoing edge $e$ of the initial location (in the automaton) a new node $\langle l, z \rangle$ is created (in the zone graph) with an edge $\langle l_0, z_0 \rangle \rightarrow \langle l, z \rangle$, where $\langle l, z \rangle$ contains the states to which the states in $\langle l_0, z_0 \rangle$ have a discrete transition through $e$. Afterwards $z$ is replaced by $z^{\uparrow}$ where $z^{\uparrow}$ denotes the set of all valuations reachable from a zone $z$ by time transitions. The procedure is repeated on every newly introduced node of the zone graph. If the states defined by a newly introduced node $\langle l, z \rangle$ are all contained in an already existing node $\langle l, z' \rangle$ ($z \subseteq z'$), $\langle l, z \rangle$ can be removed, and the incoming edge should be redirected to $\langle l, z' \rangle$.

> **Example 2.11**   For ease of understanding the algorithm is demonstrated on the automaton in Figure 2.6. The initial state is $\langle S0, z_0 \rangle$ where $z_0$ is a zone containing only the initial valuation $v_0 \equiv 0$. The initial node is $\langle start, z_0^{\uparrow} \rangle$, where $z_0^{\uparrow}$ contains all states reachable form the initial state by delay. Since as time passes, the values of the three clocks will be incremented by the same value, $x$, $y$ and $z$ has the same value in each valuation contained by $z_0^{\uparrow}$. Since there is no invariant in location $S0$ the clocks can take any positive value. Because of this $z_0$ can be defined by the constraint $x = y = z$ (that is, $x - y = 0 \wedge y - z = 0$), and the initial node can be defined as $\langle S0; x = y = z \rangle$.
>
> There is only one outgoing transition from the initial location and that resets $z$, resulting in the zone defined by $x = y \wedge z = 0$, which transforms into $z \leq x = y$ when delay is applied. This means the next node of the graph can be defined as $\langle S1, z \leq x = y \rangle$. There is only one outgoing transition from the location $S1$ and it has guard $y > 2$. This means the transition is only enabled in the subzone $z \leq x = y > 2$ (that is $z \leq x \wedge x = y \wedge y > 2$). The transition resets $y$ resulting in the zone $y = 0 \wedge z \leq x > 2$. Delay can be applied and the next node of the graph turns out to be $\langle S2, z \leq x \wedge y \leq z \wedge x - y > 2 \rangle$.
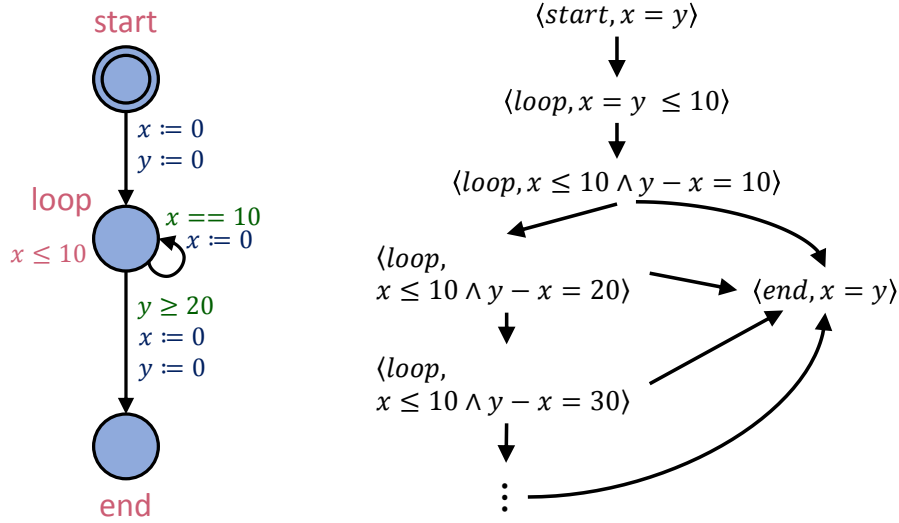
**Figure 2.7**   Timed automaton with infinite zone graph

There outgoing transition from location $S2$ has a guard $x < z + 1 \land z < y + 1$ from which $x < y + 2$ can be derived contradicting the atomic constraint $x - y > 2$ in the reachable zone of location $S2$. Thus the transition is never enabled, and location $S3$ is unreachable.

The zone graph of this automaton can be drawn as follows.

$$\langle S0; x = y = z \rangle \rightarrow \langle S1, z \leq x = y \rangle \rightarrow \langle S2, z \leq x \land y \leq z \land x - y > 2 \rangle$$

Unfortunately, it is possible that the graph described by the previous algorithm becomes infinite.

**Example 2.12**   Consider for example the automaton from [1] in Figure 2.7. Constructing the zone graph of this automaton starts similarly, with the node $\langle start, x = y \rangle$. After that both $x$ and $y$ are reset resulting in the zone defined by $x = y = 0$. Location $loop$ has an invariant $x \leq 10$ that limits the applicable delay to 10, resulting in $\langle loop, x = y \leq 10 \rangle$, where only the loop-transition is enabled.

The transition resets $x$ resulting in $\langle loop, x = 0 \land y = 10 \rangle$. Still only 10 units of delay is enabled, resulting in the node $\langle loop, x \leq 10 \land y - x = 10 \rangle$.

From this node, both transitions are enabled. The loop transition increases the difference between $x$ and $y$ yielding the new node $\langle loop, x \leq 10 \land y \leq 30 \land y - x = 20 \rangle$, while the other transition resets both clocks, resulting in the new node $\langle end, x = y \rangle$.

As we take the new node containing the location $loop$, and apply the loop

transition over and over, a new node is always constructed with the difference growing. On the other hand, the other transition always results in $\langle end, x = y \rangle$. Hence the (infinite) zone graph in Figure 2.7.

In order for the zone graph to be finite, a concept called *normalization* is introduced in [1].

Let $k(c)$ denote the greatest value to which clock $c$ is compared in the automaton. For any valuation $v$ such that $v(c) > k(c)$ for some $c$, each constraint in the form $c > n$ is satisfied, and each constraint in the form $c = n$ or $c < n$ is unsatisfied, thus the interval $(k(c), \infty)$ can be used as one abstract value for $c$.

Normalization is performed on $z^{\uparrow}$ (before inclusion is checked) in two steps. The first step is removing all constraints of the form $x < m, x \leq m, x - y < m, x - y \leq m$ where $m > k(x)$ (so that $x$ doesn't have an upper bound), and the second step is replacing constraints of the form $x > m, x \geq m, x - y > m, x - y \geq m$ where $m > k(x)$ by $x > k(x), x \geq k(x), x - y > k(x), x - y \geq k(x)$ respectively (to define the new lower bounds).

**Example 2.13**    In the automaton depicted in Figure 2.7, $k(y) = 20$ (and $k(x) = 10$). This means the exact value of $y$ doesn't really matter, as long as it is greater than 20 – the automaton will behave the exact same way if it is between 30 and 40, or if it is between 40 and 50. If we take this into consideration when constructing the zone graph, the zone $x \leq 10 \wedge y - x = 30$ can be normalized. In this zone, $y \geq 30 > k(y) = 20$, but $x \leq k(x)$. This means we only have to consider constraints bounding $y$. Implicitly $y \leq 40$ and $y - x \leq 30$. These constraints have to be removed from the zone. Similarly, $y \geq 30$ and $y - x \geq 30$ have to be replaced by $y \geq 20$ and $y - x \geq 20$. The resulting zone is $x \leq 10 \wedge y \geq 20 \wedge y - x \geq 20$. If we replace the original zone $x \leq 10 \wedge y - x = 30$ by this zone, and continue constructing the zone graph, the resulting graph is depicted in Figure 2.8.

Using normalization the zone graph is finite, but unreachable states may appear in it. If the automaton doesn't have any guard or invariant of the form $c_1 - c_2 < n$, the reachability of the location in question will be answered correctly. Otherwise, the algorithm may terminate with a false positive result.

**Example 2.14**    To demonstrate the incorrectness of the algorithm, consider again the automaton in Figure 2.6. Recall that the reachable states of the automaton (by our calculations) were $\langle S0, x = y = z \rangle$, $\langle S1, z \leq x = y \rangle$ and $\langle S2, z \leq y \leq z \leq y \wedge x - y > 2 \rangle$ – $S3$ is unreachable. Applying normalization leaves the states $\langle S0, x = y = z \rangle$ and $\langle S1, z \leq x = y \rangle$ unchanged, but the normalizing the reachable state in $S2$ results in $\langle S2, z \leq y \leq z \leq y \wedge x - y > 1 \rangle$, where the guard can be satisfied, thus making $S3$ reachable.
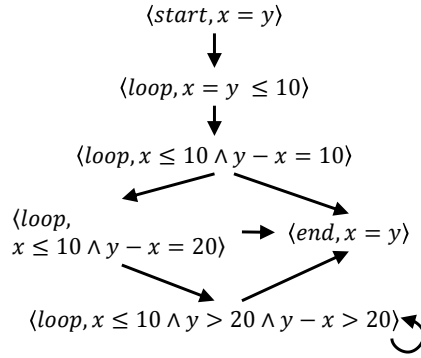
$$\langle start, x = y \rangle$$

$$\downarrow$$

$$\langle loop, x = y \leq 10 \rangle$$

$$\downarrow$$

$$\langle loop, x \leq 10 \land y - x = 10 \rangle$$

$$\langle loop, \quad x \leq 10 \land y - x = 20 \rangle \quad \rightarrow \quad \langle end, x = y \rangle$$

$$\langle loop, x \leq 10 \land y > 20 \land y - x > 20 \rangle$$

**Figure 2.8**  Finite zone graph

The operation *split* [1] is introduced to assure correctness. Instead of normalizing the complete zone, it is first split along the difference constraints, then each subzone is normalized, and finally the initially satisfied constraints are reapplied to each normalized subzone. The result is a set of zones (not just one zone like before), which means multiple new nodes have to be introduced to the zone graph (all with edges representing the same transition from the original node).

> **Example 2.15**   To demonstrate the effects of split, let us construct the zone graph of the automaton of Figure 2.6. The original node remains $\langle S0, x = y = z \rangle$, but the next node is first split along the difference constraint $x - z < 1$. Instead of the node $\langle S1, z \leq x = y \rangle$, this time there are two nodes: $\langle S1, x = y \land x - z < 1 \rangle$ and $\langle S1, x = y \land x - z \geq 1 \rangle$.
>
> From $\langle S1, x = y \land x - z < 1 \rangle$, $\langle S2, x - z \leq 1 \land z - y \leq 1 \rangle$ is reachable, where the transition to location $S3$ is not enabled because of the guard $x - z < 1$.
>
> From $\langle S1, x = y \land x - z \geq 1 \rangle$ the resulting zone after firing the transition is split along the constraint $z - y < 1$, resulting in nodes $\langle S2, x - z \geq 1 \land z - y < 1 \rangle$, and $\langle S2, x - z \geq 1 \land z - y \geq 1 \rangle$. The transition to $S3$ is not enabled in either nodes.

Applying split results in a zone graph, that is a correct and finite representation of the state space [1].

### Implementation

Paper [1] also provides an implementation of the zone domain, called *Difference Bound Matrix*, or DBM for short. The idea of DBMs is based on transforming clock constraints to difference logic formulae.

Difference constraints are easy to transform as $c_1 - c_2 \geq n$ is equivalent to $c_2 - c_1 \leq -n$ (same goes for strict inequalities), and $c_1 - c_2 = n$ is equivalent to $c_1 - c_2 \geq n \bigwedge c_1 - c_2 \leq n$. In order to transform constraints of the form $x < n$ or $x \leq n$, $x \in \mathcal{C}, n \in \mathbb{Z}$ a new variable has to be introduced.

> **Definition 2.14**   The variable denoted by **0** is a special variable that has a constant value of 0. **0** is not a clock variable, but can appear in clock constraints.

Using **0** $x \sim n$, $\sim \in \{\leq, <, =, >, \geq\}$ can be transformed into $x - \mathbf{0} \sim n$, and all clock constraints can be transformed into the desired form.

> **Definition 2.15**   A *Difference Bound Matrix D* of a zone $z$ operating on $\mathcal{C}$ is a square matrix of $|\mathcal{C}| + 1$ rows (and columns). A row and a column is assigned to each $c \in (\mathcal{C} \cup \{\mathbf{0}\})$. Each element $D_{i,j}$ of the matrix describes an upper bound on $i - j$, by storing whether the inequality is strict ($<$ or $\leq$) and the bound $n$. It is possible that there is no upper bound on $i - j$, in this case $D_{i,j} = \infty$.
> The DBM $D$ of zone $z$ stores all constraints bounding $z$.

In order for operations to be effective it is required that the DBM is in a *canonical* form.

> **Definition 2.16**   A Difference Bound Matrix $D$ of a zone $z$ is in canonical form if for all $i, j \in (\mathcal{C} \cup \{\mathbf{0}\})$, $D_{i,j}$ denotes the strictest bound on $i - j$ that can be derived from $z$.

As zones and DBMs are different representations of the same entity, this paper uses the terms interchangeably.

Many operations are defined on DBMs. The most important ones are the following:

**consistent($D$)**   is used to decide if $D$ contains any states

**relation($D, D'$)**   tells if one of $D$ and $D'$ are contained in the other

**satisfied($D, m$)**   where $m$ is a difference constraint tells if $D$ contains any states satisfying $m$ without affecting $D$

**up($D$)**   calculates $D^{\uparrow}$

**and($D, m$)**   where $m$ is a difference constraint restricts $D$ to the states satisfying $m$

**free($D, c$)**   where $c \in \mathcal{C}$ removes all constraints on $c$

**reset($D, c$)**   where $c \in \mathcal{C}$ resets $c$

**norm(D, k)**  where $k : \mathcal{C} \to \mathbb{Z}$ normalizes the zone based on $k$ that assigns to each
   $c \in \mathcal{C}$ the highest value they are compared to in an automaton

**split(D, $\mathcal{G}$)**  where $\mathcal{G} \in \mathcal{B}(\mathcal{C})$ splits the zone based on $\mathcal{G}$ that is the set of all difference
   constraints appearing in an automaton.

Using these operations the calculation of the next zone on the zone graph can
be automated. Let $n = (l, D)$ be an already calculated node of the zone graph, and
$e = (l, g, r, l')$ an edge of the timed automaton. Calculating the next node $n' = (l', D)$
(or next nodes) starts by checking guards. This can be performed by calling *and(D,m)*
for each atom $m$ of the difference logic formula representation of $g$ and then checking
*consistent(D)* to check if there are any states in $D$ satisfying $g$. If there are, the transition
is enabled. In this case $D$ has to be reset: for $c \in r$, *reset(D,c)*. $I(l')$ also has to be
satisfied in order for the transition to be enabled. This can be checked similarly to the
guard. After this *up* can be used to calculate $D'$, but *and(D,m)* has to be called again for
$m \in I(l')$. After that *split* and *norm* are called to ensure correctness and termination.

Implementations (pseudocodes) of these operations are provided in [1]. Termina-
tion of the algorithm is also proven, but it's complexity is exponential in the number of
clocks. Because of this it is essential to reduce the number of clocks as much as it is
possible, without changing the reachability property.

### Activity

In [4] abstractions of the automaton are proposed to reduce the number of clock
variables without affecting the operation of the automaton. The abstraction that will be
used later in this paper is called *activity*. A clock $c$ is considered active at some location
$l$ (denoted by $c \in Act(l)$) if its value at the location may influence the future evolution
of the system. It might be because the clock appears in the invariant of the location, or
in the guard of some outgoing edges of the location, or because it is active in one of the
posterior locations and its value is not reset until that location.

**Example 2.16**   In the automaton depicted on figure 2.6 clock $z$ is active at location
$S2$ because it appears in the guard of the outgoing edge. It is also active in $S1$ because
its value in $S1$ determines its value in $S2$ and it is active in $S2$, but it is not active in
$S0$ because its value is not important, since it is reset in the outgoing edge anyway.

The core of the algorithm for reducing the number of clock variables is to calculate
$Act(l)$ for each $l \in L$, and if $Act(l) < |\mathcal{C}|$ holds for each $l \in L$, the automaton can be
reconstructed by *renaming* variables location by location (after renaming there will be
less clocks). This is true, even if all $c \in \mathcal{C}$ is active in at least one location, however,
clocks might be renamed differently in distinct locations.

Before presenting how activity is calculated some new notations are introduced. Let $clk : \mathcal{B}(\mathcal{C}) \to 2^{\mathcal{C}}$ and assign to each clock constraints the set of clocks appear in it. Define $clk : L \to 2^{\mathcal{C}}$ such that $c \in clk(l)$ iff $c \in clk(I(l))$ or there exist an edge $(l, g, r, l')$ such that $c \in clk(g)$.

*Activity* is calculated by an iterative algorithm starting from $Act_0(l) = clk(l)$ for each $l \in L$. In the $i^{th}$ iteration $Act_i(l)$ is derived by extending $Act_{i-1}(l)$ by $Act_{i-1}(l') \setminus r$ for each edge $(l, g, r, l')$. The algorithm terminates when it reaches a fix point, i.e. when $Act_i(l) = Act_{i-1}(l)$ for each $l \in L$.

> **Example 2.17**    Let us calculate activity for the complete automaton. $Act_0(l) = clk(l)$ for each $l \in L$. Thus, iteration starts from $Act_0(S0) = \emptyset$, $Act_0(S1) = \{y\}$, $Act_0(S2) = \{x, y, z\}$, and $Act_0(S3) = \emptyset$.
>
> Since $y$ is not reset on edge $S0 \to S1$, $Act_1(S0) = \{y\}$. Since $x$ and $z$ are not reset on edge $S1 \to S2$, $Act_1(S1) = \{y, x, z\}$. Since $Act_0(S3) = \emptyset$, and $S3$ has no outgoing edges $Act_1(S2) = \{x, y, z\}$ and $Act_1(S3) = \emptyset$.
>
> Clock $x$ is not reset on edge $S0 \to S1$ but $z$ is, thus $Act_2(S0) = \{y, x\}$. Other activities are unchanged in this iterations, thus $Act_2(S1) = \{y, x, z\}$, $Act_2(S2) = \{x, y, z\}$, $Act_2(S3) = \emptyset$ and the fix point is reached.

### Complexity

As it was mentioned, reachability for timed automata without discrete variables is decidable (but it it exponential). It was also mentioned before that reachability for finite automata extended by (discrete) variables is undecidable. Obviously, reachability of timed automata extended with discrete variables is also undecidable. However it is decidable if the value sets of the discrete variables are finite, because in this case the values can be encoded in the locations.

## 2.4   Objectives

The goal of this paper is to provide an extendable framework for CEGAR-based algorithms deciding reachability of timed automata extended with discrete variables. Since reachability is undecidable for this type of timed automata, termination of the algorithms is not always guaranteed.
.

Chapter 3

# Configurable Timed CEGAR

This chapter presents a configurable framework for CEGAR-based reachability analysis of timed automata.

## 3.1  Generic CEGAR Framework

The key idea of the framework is to provide various implementations of each phases of the CEGAR-loop, by using correspondent parts of CEGAR-based reachability algorithms. Most of these algorithms already exist (mostly for other formalisms, and they have to be adapted to timed automata), but some of them are new approaches. The implemented modules can then be combined (that is, the implementation of each phases can be provided by different algorithms) to form new algorithms, and choosing the most effective parts of the original algorithms can result in an even more effective algorithm then the original ones.

The architecture of the framework is illustrated on figures 3.1 and 3.2. There are two different realizations of the CEGAR-loop, because the framework supports two distinct ways to apply abstraction to timed automata. While the first approach (figure 3.1) is based on eliminating clock variables (see section 2.2.3): starting from a finite automaton (without any clock variables) the current automaton gets extended with some clocks in each iteration - and the *automaton* is the base of refinement, the other (figure 3.2) is based on the refinement of the *statespace* itself. Only implementations of the same approach are interchangeable – hence the two realizations.

### 3.1.1  Automaton-based refinement

Figure 3.1 depicts the architecture of the automaton-based approach. The initial abstraction is a finite automaton that is derived from the original timed automaton by removing all clock variables and clock constraints.
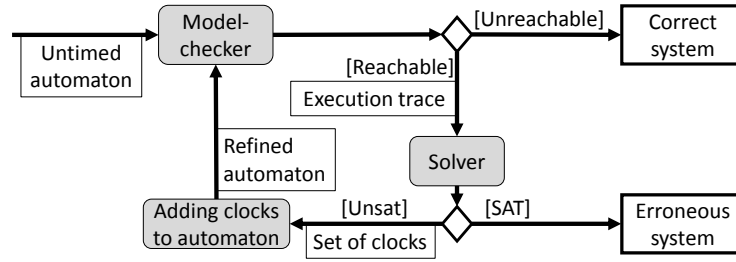
**Figure 3.1**    Automaton-based refinement

In each iteration of the CEGAR-loop, the task of the model checking phase is to determine whether the error location is reachable in the current automaton and provide a trace (counterexample) if there is one. Therefore, the implementation should be a reachability-checking algorithm for timed automata that can find a trace to the location.

The task of the analysis phase is to check if the found trace is feasible in the original automaton and if it isn't, provide a set of clock variables that can then be added to the automaton (with the clock constraints they appear in) so that the model checker won't find this counterexample again. This can be implemented by a solver.

Finally, the only task of the refinement phase is to refine the current abstraction of the automaton, by extending it with the given set of clock variables (and the constraints they appear in). The task is straightforward, and so this part of the CEGAR-loop has only one implementation.

**TODO:** A pseudocode is provided to demonstrate implementability.

### 3.1.2   Statespace-based refinement

In case of statespace-based refinement, the representation of the statespace has a defining role. In the proposed framework, the statespace is represented by zone graphs - this is common for all algorithms. However, the abstraction of the zone graph can be
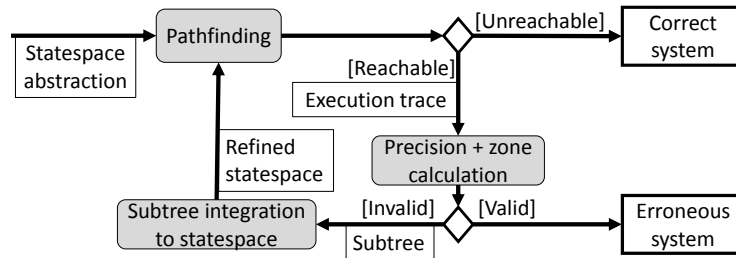


**Figure 3.2**    Statespace-based refinement

performed various ways. In this framework, the main idea is to explore the statespace without considering clock variables (and in some cases discrete variables, too), and to refine the statespace - trace by trace - by deciding which of the clock variables to include for each of the zones on that path. Afer that the graph is refined (clocks are included in the zones), and during the refinement it turns out whether the counterexample is feasible or not. Figure 3.2 depicts the architecture of this approach.

Because of the different approaches of abstraction, consructing the initial abstraction is not as straightforward as it was in case of automaton-refinement. All that can be said is that it is some sort of abstraction of the statespace derived from the automaton without including clock variables.

The task of the model checking phase is to find a path from the initial location to the error location in the current abstraction of the zone graph. Because of this, the model checking phase of statespace-based refinement is performed by pathfinding algorithms.

The task of the analysis phase can be divided into two parts: decide which of the clock variables to include in the zones (i.e. the *precision* of the zone) and calculate the zones on the trace (up to the given precision) and find out if it is feasible or not.

When performing the first part it is important to find precisions (that might change along the trace) that is not too big (does not include too much variables) to be calculated efficiently, but includes all variables that are necessary to find out if he trace is feasible or not. The result of this part should be a function $P : V(G) \rightarrow 2^{\mathcal{C}}$ assigning precisions to the nodes of the current abstraction of zone graph.

As for the second part, calculating the correct zones can be performed by the steps of the algorithm presented in section 2.3.2 with some modifications that help with handling the changes of precision along the zones in counterexample. If the error location is unreachable, a guard or invariant will eventually prove to one of the edges on the trace that it represents a transition that is not enabled.

The task of the refinement phase is to modify the current abstraction of the zone graph according to the states calculated in the analysis part.

**TODO:** A pseudocode is provided to demonstrate implementability.

The presented methods for model checking and refinement describe the essence of the algorithms that seem to be the same, however, the concrete implementation depends of the structure of the abstract zone graph representation. Because of this only those modules can be used interchangeably, that are defined for the same representation.

## 3.2  Modules

This section describes the implementations of the previously defined interfaces. Methods for automaton-based refinement are explained: model checkers, an algorithm for calculating the set of clock variables to refine the automaton, and the general algorithm for performing the refinement. Two possible representations of the abstract zone graph

are presented. Algorithms for statespace-based CEGAR are shown, mentioning the statespace representation-dependent behaviours of model checking and refinement.

Algorithms are presented on the automaton on figure 2.7 with $l_{err} = end$ to ease understanding. From now on $\mathcal{A}$ will refer to that particular timed automaton.

### 3.2.1  Zone graph exploration
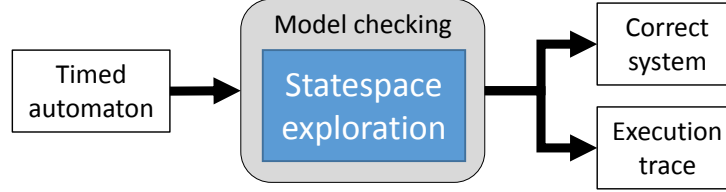


The reachability-checking algorithm described in part 2.3.2 is an obvious choice for the model checking phase, however, it is important to note that the algorithm does not handle discerete variables. The discrete valuation can be encoded into the location (and calculated on the fly) but in this case termination is not ensured (as part 2.2.1 explains).

### 3.2.2  Satisfiability-based model checker
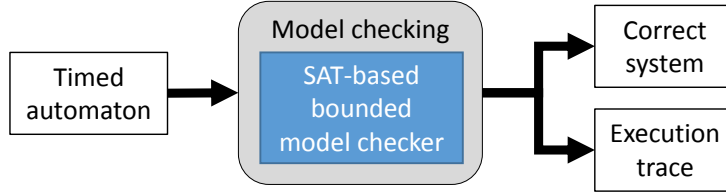


Satisfiability-based model checking as introduced in section 2.2.3 can be directly applied to timed automata – the only necessary change is to define a transformation that can turn a counterexample (an execution trace) into a SAT-problem.

The idea is to separate discrete transitions from time transitions. Consider a counterexample sequence $\sigma = l_0 \xrightarrow{t_0} l_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} l_{err}$. This representation of $\sigma$ hides the fact that it is important how much time the systems spends in each locations - i.e. delay transitions. Let us denote the amount of time spent in $l_i$ by $d_i$. This way $\sigma$ can be defined by $\sigma = l_0 \xrightarrow{d_0}\xrightarrow{t_0} l_1 \xrightarrow{d_1}\xrightarrow{t_1} \cdots \xrightarrow{d_n}\xrightarrow{t_n} l_{err}$. In this representation $\xrightarrow{d_i}$ can be considered a special kind of transition that increases $v(c)$ for each $c \in \mathcal{C}$ by $d_i$. Based on this the SAT formula can be constructed.

First, let us assign a variable for each clock in each location, both before and after the delay – that is, this means $2 \cdot n \cdot |\mathcal{C}|$ variables. Let us denote these variables by $c_i$ (for

the value of clock $c$ in location $l_i$ before the delay) and $c_i'$ (for the value of clock $c$ in location $l_i$ after the delay). Let us also assign variables for each $d_i$. The first constraints that have to be added is that each of the defined variables are greater or equal to 0.

The initial constraints can simply be described by $c_0 = 0$ for each $c \in \mathcal{C}$. Delay transitions can be turned into constraints by the following equation $c_i + d_i = c_i'$ for each $c \in \mathcal{C}, 0 \leq i \leq n$. In case of discrete transitions, guards (clock constraints) can be turned into solver constraints by replacing the clock variables with the defined variables. The guard $g_i$ of a transition $t_i(l_i, g_i, r_i, l_{i+1})$ can be transformed by replacing all clocks $c$ appearing in $g_i$ by $c_i'$. Resets can also be simply transformed into constraints – for all $c \in r_i$ $c_{i+1} = 0$ has to be added to the set of constraints. Note, that this way $c_{i+1}$ is only specified for the reset clocks. For all $c \notin r_i$ $c_{i+1} = c_i'$ has to be added to the set of constraints. Invariants can be transformed into solver constraints the same way as guards.

Discrete variables can be mapped to solver variables as before since discerete variables and clock variables have no affect on eachother.

> **Running example 3.1** Consider automaton $\mathcal{A}$. Assume the counterexample is $\sigma$ = emphstart → emphloop → emphend. The new variables are $x_0, y_0, x_0', y_0', x_1, y_1, x_1', y_1', x_2, y_2, x_2', y_2', d_2, d_1$ and $d_2$. It is important to define them to be non-negative since they represent the elapse of time. Initial constraints are $x_0 = 0$ and $y_0 = 0$. Delay is described by constraints $x_0' = x_0 + d_0$, $y_0' = y_0 + d_0$, $x_1' = x_1 + d_1$, $y_1' = y_1 + d_1$, $x_2' = x_2 + d_2$ and $y_2' = y_2 + d_2$.
>
> There is no guard on edge *start* → *loop*, but it resets both variables yielding the constraints $x_1 = 0$, $y_1 = 0$. The invariant of location *loop* can be transformed to $x_1 \leq 10$ and $x_1' \leq 10$. The next edge has a guard, $y \geq 20$ that can be transformed to $y_1' \geq 20$, and the resets to $x_2 = 0$ and $y_2 = 0$. This results in the formula $\varphi = x_0 = 0 \bigwedge y_0 = 0 \bigwedge x_0' = x_0 + d_0 \bigwedge y_0' = y_0 + d_0 \bigwedge x_1' = x_1 + d_1 \bigwedge y_1' = y_1 + d_1 \bigwedge x_2' = x_2 + d_2 \bigwedge y_2' = y_2 + d_2 \bigwedge x_1 = 0 \bigwedge y_1 = 0 \bigwedge x_1 \leq 10 \bigwedge x_1' \leq 10 \bigwedge y_1' \geq 20 \bigwedge x_2 = 0 \bigwedge y_2 = 0$.

This allows us to use a SAT-solver to decide if a possible execution trace of a timed automaton is feasible. This can be used for model checking timed automata, by iterating over all possible execution traces and if a trace $\sigma$ is found from $l_0$ to $l_{err}$, it can be checked, and if the derived formula is satisfiable, $\sigma$ is proposed as a counterexample.

The problem with this model checker is that there may be infinitely many execution traces. Thus, this model checker can only be used as a *bounded* model checker.

**TODO:** A pseudocode is provided to demonstrate implementability.
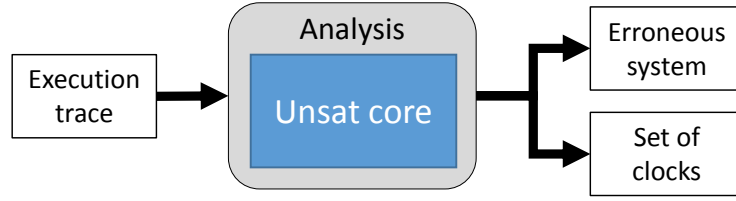
### 3.2.3   Unsat core-based clock selection

Solvers can be useful, not only to decide if a given set of constraints is satisfiable, but also - if the answer is that the formula is unsatisfiable - solvers have various features to show why they can not be satisfied. One of the possible helpful feature is deriving the so called *unsat core* - that is, a minimal set of the given constraints that is unsatisfiable in itself. This set of constraints can be used to determine the set of clock variables with what the current abstraction of the automaton has to be extended. In order to define the refinement set, the variables appearing in the unsat core have to be transformed back to the original variables. The set of original variables appearing in the constraints is the result of the algorithm.

> **Running example 3.2**   The result of checking satisfiability of $\varphi$ is *UNSAT*, and the unsat core is $\varphi_{unsat} = x_1' = x_1 + d_1 \bigwedge y_1' = y_1 + d_1 \bigwedge x_1 = 0 \bigwedge y_1 = 0 \bigwedge x_1 \leq 10 \bigwedge x_1' \leq 10 \bigwedge y_1' \geq 20$. This means the automaton has to be refined with both $x$ and $y$ so that the spurious counterexample is eliminated.

### 3.2.4   Automaton refinement
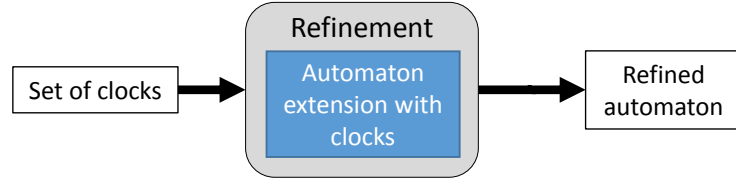


Given an original automaton $\mathcal{A}$ an abstract automaton $\mathcal{A}'$ and a set of clock variables to be added $C \subseteq \mathcal{C}$, the task is to refine $\mathcal{A}'$ so that each clock $c \in C$ appears in it. The task is to decide which of the guards, resets and invariants to include. Resets are easy to add: the ones that reset clocks in $C$ should be included, others don't. Guards and invariants are clock constraints – conjunctive formulae of atomic constraints bounding the value of the clocks or the difference of two clocks. Decision can be made for each atomic formula one by one: those in which only clocks in $\mathcal{A}'$ or $C$ appear – that is, difference constraints are only included if both clocks appear in $\mathcal{A}'$ or $C$.

$$\langle start, z_\emptyset \rangle$$

$$\langle loop, z_\emptyset \rangle$$

$$\langle end, z_\emptyset \rangle$$
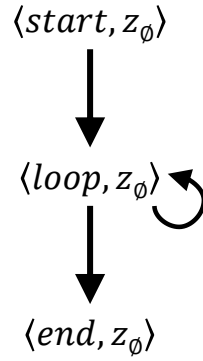
**Figure 3.3**    Initial abstraction of $\mathcal{A}$

### 3.2.5   Graph representation

The first representation of the abstract zone graph is another zone graph, with zones of varied precisions. To avoid confusion, from now on precisions of zones will always be shown: zones will be denoted by $z_C$ where $C \subseteq \mathcal{C}$ is the precision of the zone. Zones of the real zone graph (without abstraction) are denoted by $z_{\mathcal{C}}$.

A node $\langle l, z_C \rangle$ of the abstract zone graph can represent any nodes $\langle l, z'_{\mathcal{C}} \rangle$ of the real zone graph, that contains the same location $l$, and some zone $z'_{\mathcal{C}}$ for which $z'_C \subseteq z_C$ (where $z'_C$ means a spatial projection of $z'_{\mathcal{C}}$ to the subspace spanned by the clocks in $C$) holds. This means $\langle l, z_\emptyset \rangle$ can represent any nodes of the real zone graph containing $l$.

Based on this the initial abstraction can be constructed by assigning a node $\langle l, z_\emptyset \rangle$ to each location $l \in L$. The graph can then be compled with edges: for each $e = (l, g, r, l') \in E$ a new edge of the zone graph should be included pointing from $\langle l, z_\emptyset \rangle$ to $\langle l', z_\emptyset \rangle$.

> **Running example 3.3**    The initial abstraction for $\mathcal{A}$ is depicted on figure 3.3.

During the algorithm this graph will be refined by the zones calculated in the refinement phase. Sometimes nodes will get replicated, or edges deleted (the precise algorithm will be described later), but it will remain to be an absraction of the real zone graph. Discrete valuations are also calculated in the refinement phase.

### 3.2.6   Tree representation

The other representation of the abstract zone graph is based on the idea of search trees. Instead of keeping track of the full (abstract) zone graph (like we did with the other representation) details of the tree will be uncovered in the model checking phase of the CEGAR loop. However, one thing is common in both representations: the abstraction

of the nodes is based on a set of clocks (precision) to include (just like in case of the automaton-based refinement) and initially all precisions are empty. The statespace exploration will also operate on empty precision sets, and the zones will be calculated in the refinement phase. In this case, discrete valuations can be calculated during statespace exploration (but it is not necessary).

Let us define the formalism to represent the abstract tree graph.

**Definition 3.1**  The auxiliary graph can be defined as a tuple $\langle N_e, N_u, E^\uparrow, E^\downarrow \rangle$ where

- $N_e \subseteq L \times \mathcal{B}(\mathcal{C})$ is the set of explored nodes,

- $N_u \subseteq L \times \mathcal{B}(\mathcal{C})$ is the set of unexplored nodes,

- $E^\uparrow \subseteq (N_e \times N)$, where $N = N_e \cup N_u$ is the set of upward edges and

- $E^\downarrow \subseteq (N_e \times N)$ is the set of downward edges.

The sets $N_e$ and $N_u$ as well as the sets $E^\uparrow$ and $E^\downarrow$ are disjoint. $T^\downarrow = (N, E^\downarrow)$ is a tree.

Nodes are built from a location and a zone like in the zone graph but in this case nodes are distinguished by their trace reaching them from the initial node. This means the graph can contain multiple nodes with the same zone and the same location, if the represented states can be reached through different traces. The root of $T$ is the initial node of the (abstract) zone graph. A downward edge $e$ points from node $n$ to $n'$ if $n'$ can be reached from $n$ in one step in the zone graph.

Upward edges are used to collapse infinite traces of the representation, when the states are explored in former iterations. An upward edge from a node $n$ to a previously explored node $n'$ means that the states represented by $n$ are a subset of the states represented by $n'$, thus it is unnecessary to keep searching for a counterexample from $n$, because if there exists one, another one will exist from $n'$. Searching for new traces is only continued on nodes without an upward edge. This way, the graph can be kept finite, unless the discrete variables of the automaton prevent it.

Initially, the graph contains only one, unexplored node $\langle l, z_\emptyset \rangle$, and as the statespace is explored, unexplored nodes become explored nodes, new unexplored nodes and edges appear, until a counterexample is found. During the refinement phase zones are calculated, new nodes and edges appear and complete subtrees disappear. Statespace exploration will then be continued from the unexplored nodes, and so on. Discrete valuation can be calculated during statespace exploration.

**Running example 3.4**  The initial abstraction for $\mathcal{A}$ is a single unexplored node $n_0 = \langle start, z_\emptyset \rangle$.

### 3.2.7   Statespace exploration
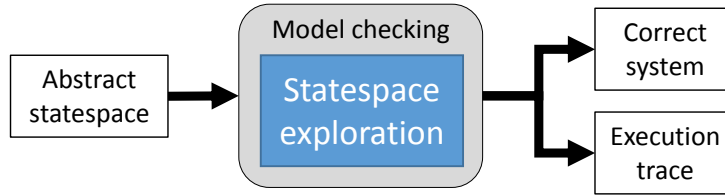


The task of the model checking phase is to find traces from $l_0$ to $l_{err}$. In case of the graph representation, where $l_{err}$ appears in the node $\langle l_{err}, z_\emptyset \rangle$ even in the initial abstraction, model checking becomes a path finding problem from $\langle l_0, z_\emptyset \rangle$ to $\langle l_{err}, z_\emptyset \rangle$ in the abstract zone graph. This can be performed by any path finding algorithm.

> **Running example 3.5**   Pathfinding in $\mathcal{A}$'s initial abstraction finds the previously mentioned counterexample $\sigma = $ emphstart $\rightarrow$ emphloop $\rightarrow$ emphend.

In case of the tree representation, $l_{err}$ does not appear in the graph and the statespace exploration has to be continued until a node $\langle l_{err}, z_\emptyset \rangle$ appears. Statespace exploration has to be performed the following way.

In each iteration a node $n = \langle l, z_C \rangle \in N_u$ for some $C$ is chosen. First, it is checked if the states $n$ represents are included in some other node $n' = \langle l, z'_C \rangle$ with a zone of the same precision. If this is the case an upward edge is introduced from $n$ to $n'$ and $n$ becomes explored. Otherwise, $n$ has yet to be explored. For each outgoing edge $e(l, g, r, l')$ of $l$ in the automaton a new unexplored node $\langle l, z_\emptyset \rangle$ is introduced with an edge pointing to it from $n$, which becomes explored. If any of the new nodes contains $l_{err}$, the algorithm terminates. Otherwise, another unexplored node is chosen, and so on.

> **Running example 3.6**   Exploration of $\mathcal{A}$'s abstract statespace starts by exploring $n_0$, that is performed by introducing a new (unexplored node) $n_1 = \langle loop, z_\emptyset \rangle$ with a downward edge $n_0 \rightarrow n_1$. The erroneous location is not explored yet so the iteration continues. Location $loop$ has two outgoing edges. The loop edge introduces $n_2 = \langle loop, z_\emptyset \rangle$ with a downward edge $n_1 \rightarrow n_2$. Edge $loop \rightarrow end$ introduces $n_3 = \langle end, z_\emptyset \rangle$ with $n_1 \rightarrow n_3$. The erroneous location is found and $\sigma = n_0 \rightarrow n_1 \rightarrow n_3$ is proposed as a counterexample.

**TODO:** pszeudokód
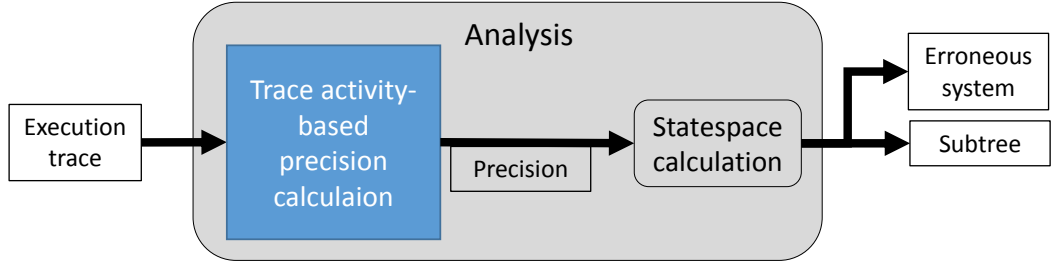
### 3.2.8   Trace Activity-based precision calculation

The task of the analysis phase is to determine the precision of each zones on a given counterexample. The abstracion *activity* as described in section 2.3.2 is able to assign a set of clocks for each locations of the automaton, without affecting its behaviour. Assigning $act(l)$ for each node $n = \langle l, z_C \rangle$ would be a good solution of the task, however it can be made more effective by considering the fact that we are only considering an execution trace, and we only need to know if it is feasible.

Based on *activity* a new abstraction can be introduced, called *trace activity* $Act_\sigma(n)$ : $N \to 2^\mathbb{C}$ which does the same thing as activity, except for a trace: it assigns precisions to nodes (not locations in this case, because the same location may appear multiple times ion a trace with different activity). The algorithm calculating trace activity operates the following way.

The algorithm iterates over the counterexample trace, but backwards. In the final node $n_{err} = \langle l_{err}, z_\emptyset \rangle$ it is not important to know the valuations, as the only important thing to know if it is reachable. Therefore $Act_\sigma(n_{err}) = \emptyset$. After that $Act_\sigma(n_i)$ can be calculated from $Act_\sigma(n_{i+1})$ and the edge $e_i(l_i, g_i, r_i, l_{i+1})$ used by transition $t_i$. Since $r_i$ resets clocks, their values in $l_i$ will have no effect on the systems behavior in $l_{i+1}$. Thus clocks in $r_i$ can be excluded. It is necessary to know if $t_i$ is enabled, so $clk(g_i)$ must be active in $n_i$. It is also important to satisfy the invariant of $l_i$ thus $clk(I(l_i))$ must be included. This gives us the formula $Act_\sigma(n_i) = (Act_\sigma(n_{i+1}) \setminus r_i) \cup clk(g_i) \cup clk(I(l_i))$.

> **Running example 3.7**   Calculating trace activity of $\sigma = n_0 \to n_1 \to n_3$ starts from the erroneous node $n_3$ by $Act_\sigma(n_3) = \emptyset$. Both $x$ and $y$ are reset on edge $loop \to end$ and only $y$ is tested, but $x$ appears in *loop*'s invariant, thus $Act_\sigma(n_1) = \{x, y\}$. Since both clocks are reset on edge $start \to loop$, $Act_\sigma(n_0) = \emptyset$.

### 3.2.9   Unsat core-based precision calculation

Unsat core can also be used to determine the necessary precision of a given counterexample. First, the SAT formula described in part 3.2.2 is checked by a solver. If it is satisfiable, the counterexample is feasible. Thus, there is no need to refine the graph,
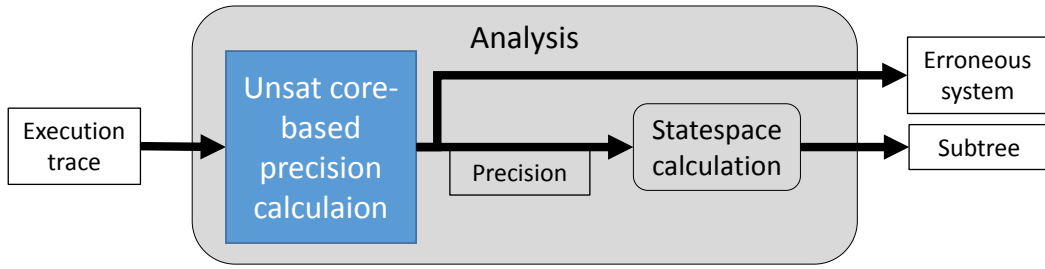
the CEGAR algorithm can terminate (or $\emptyset$ can be assigned to all nodes as a precision and the algorithm will terminate in the refinement phase). Otherwise, unsat core has to be examined. When constructing the SAT formula, variables were introduced for each step. Thus precision can be obtained from the unsat core by step: if $c_i$ or $c_i'$ appears in the unsat core $c$ must be included in the precision assigned to $n_i$.

> **Running example 3.8**  As it was mentioned, the unsat core of $\varphi$ is $\varphi_{unsat} = x_1' = x_1 + d_1 \bigwedge y_1' = y_1 + d_1 \bigwedge x_1 = 0 \bigwedge y_1 = 0 \bigwedge x_1 \leq 10 \bigwedge x_1' \leq 10 \bigwedge y_1' \geq 20$. The included variables representing clock variables are $x_1', x_1, y_1'$ and $y_1$. This results in precision $\emptyset$ for $n_0$, $\{x, y\}$ for $n_1$ and $\emptyset$ for $n_3$.

### 3.2.10  Statespace calculation
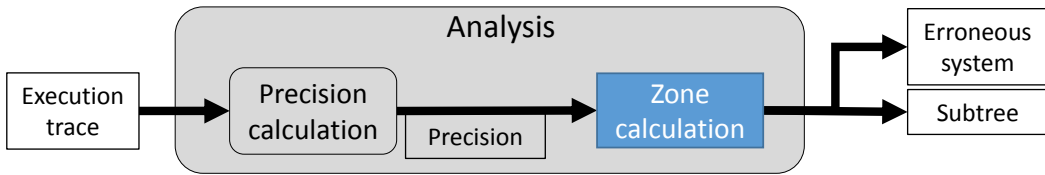


The task of the refinement phase is to assign correct zones of the given precision for each node in the trace. It is important to mention that the zones on the trace may already be refined to some precision $C'$ that is independent from the new precision $C$. In this case the zone has to be refined to the precision $C \cup C'$. The initial zone can be calculated as described in part 2.3.2, except this time not all variables have to be included. After that for each edge in the trace, the zone in the next node can be calculated with some little modifications of the corresponding part of the zone graph exploration algorithm regarding the precision change.

Assume the zone $z_i$ of node $n_i$ is refined to precision $C_i$ and the next zone $z_{i+1}$ in node $n_{i+1}$ has to be refined to $C_{i+1}$. Consider the DBM implementation of zones. Variables $C_{old} = C_i \setminus C_{i+1}$ have to be excluded from the precision. This can be done by performing *free(c)* for each $c \in C_{old}$, but in [1] the operation *free(c)* only affects

the row and the column belonging to $c$. Thus, for space saving purposes, the row and column of $c$ can simply be deleted from the DBM.

Variables $C_{new} = C_{i+1} \setminus C_i$ have to be introduced. This is a more complex task, since the value is necessary to know. *Trace activity* is constructed in a way that new clocks can only appear when they are reset. In this case, introducing the new variable is simple: add a new row and column to the DBM, belonging to $c$ and call *reset(c)*. However this is not always the case for *unsat core*. It is possible that some constraints only appear in the unsat core, because they contradict each other, or a variable $c$ may appear in the unsat core, because several constraints combined can result in an unsatisfiable constraint that does not include $c$.

> **Example 3.9**  Consider the automaton on figure 2.6. The unsat core-based precision of $S1$ is $\{x, y\}$, and $\{x, y, z\}$ of $S2$, but $z$ is not reset on $S1 \rightarrow S2$. In the unsat core it appears in $x < z + 1$ and $z < y + 1$ that imply $x < y + 2$.

It is clear that in this case the concrete value of the variable $z$ doesn't matter, it is only there so that the constraints it appears in are considered. Because of this, there is no need to assign a precise value to $z$ - introduce a row and a column belonging to $z$ and then call *free(z)*.

The correct zones on the trace are calculated. It is important to consider that sometimes the *split()* operation results in more than one zones. In this case the corresponding node is replicated and one of the result zones is assigned to each versions of the node. Exploration has to be continued from that node, thus the refinement of a trace may result in a tree.

> **Running example 3.10**  The refinement of trace $\sigma$ results in $\langle start, z_\emptyset \rangle \rightarrow \langle loop, x = y \leq 10 \rangle$.

**TODO:** pszeudokód az eddigiekről

### 3.2.11   Statespace refinement
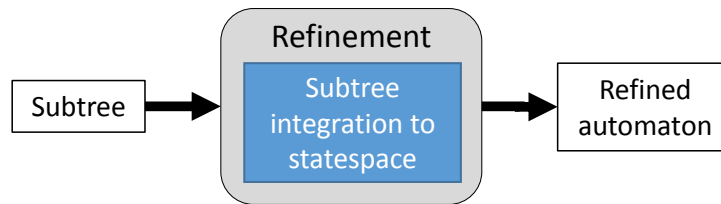


The next important question is how to integrate the refined tree to the graph. The answer depends on which representation is used.

In case of the graph representation integrating has to be done carefully. Before changing the abstract zone to the refined one we must consider the other incoming edges of the node. The states reachable from that edge may not be contained in the refined zone, and thus if there is an an edge pointing to the node to refine other than the one in the trace, the node should be duplicated, and the other incoming edges should be pointing to the new node (that doesn't get refined). Also, if the result of *split()* is multiple zones, the node has to be replicated, but this time no edges has to be redirected, and one of the refined zones can be assigned to each nodes.

Discrete valuation also has to be calculated at this point. The same discrete valuation has to be assigned for each replicas of the node.

The next step is checking containment. Suppose at one point of the algorithm the zone $z_C$ in node $n$ is refined to $z_{C'}$ which is a subzone of a zone $z'_{C'}$ in a node $n'$ containing the same location. In this case any state that is reachable from  is also reachable from $n'$, thus any edge leading to $n$ can be redirected to $n'$, and $n$ can be removed.

If the erroneous location is reachable through this path, the procedure finds it, and the CEGAR algorithm terminates. Otherwise, at some point a guard or a target invariant is not satisfied – the transition is not enabled. The corresponding edge is removed and the analysis of the path terminates.

> **Running example 3.11**    Integration of the refined trace to the abstraction of the zone graph starts from the first node. Since the zone is still $z_\emptyset$, there is not much to be done. The next node on the trace is $\langle loop, z_\emptyset \rangle$. The loop edge is an incoming edge to the node itself. Thus, the node is duplicated and the loop-edge from the original one is redirected to the new one. Now the zone in the original node can be refined to $x = y \leq 10$. Since $loop \to end$ is not enabled on the trace, the outgoing edge from $\langle loop, x = y \leq 10 \rangle$ is removed from the graph.
>
> The graph after the process looks as follows.
>
> $$\langle start, z_\emptyset \rangle \to \langle loop, x = y \leq 10 \rangle \to \langle loop, z_\emptyset \rangle \to \langle end, z_\emptyset \rangle$$

**TODO:** pszeudokód?

Incoming edges that are not on the trace are also important in case of tree representation, however, because of the tree nature of $T$, the other incoming edges of a node $n$ can only be upwards edges, representing that all states represented by some node $n'$ are also represented by $n$. Obviously, this may not be true, after refining the zone in the node, and because of this the edge $n' \to n$ is removed, and $n'$ is marked as unexplored.

Since $T$ is already a tree, it does not cause problems to attach new subtrees to it (because of *split*), but all new nodes have to be marked as unexplored, since only one outgoing edge (of the automaton) were considered when calculating the new subtree,

and there could be more.

   Containment can also be checked here, just as in case of the graph representation, but it only matters for the leaves of the tree (since the other nodes are already explored). The other possibility is to mark the leaves unexplored and statespace exploration will search for containment.

> **Running example 3.12**   Since there are no upwards edges in the current tree, refinement can be performed by replacing the zone in $n_1$ with the refined one and removing edge $n_1 \rightarrow n_3$ resulting in the following graph.
>
> $$\langle start, z_\emptyset \rangle \rightarrow \langle loop, x = y \leq 10 \rangle \rightarrow \langle loop, z_\emptyset \rangle$$

**TODO:** pszeudokód?
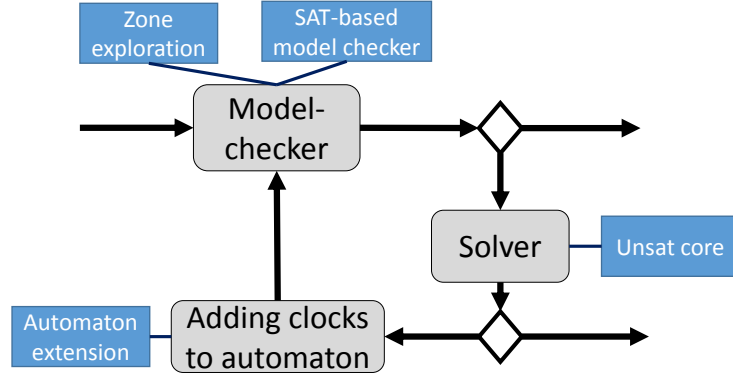
## 3.3   Result



**Figure 3.4**   Automaton-based refinement

   Fiugre 3.4 depicts the presented implementations for automaton-based refinement and figure 3.5 depicts the presented implementations for statespace refinement. In the latter case statespace representation-dependent implementations are marked with the same colors. Precision and zone calculation are not statespace representation-dependent algorithms and can be combined with anything.

   The presented framework is extendable in may ways. New implementations of each modules can be added to the framework and combined with existing ones. New representations can be defined for statespace-based refinement, and even a new aspect of abstraction can be introduced with it's own CEGAR-loop implementation.
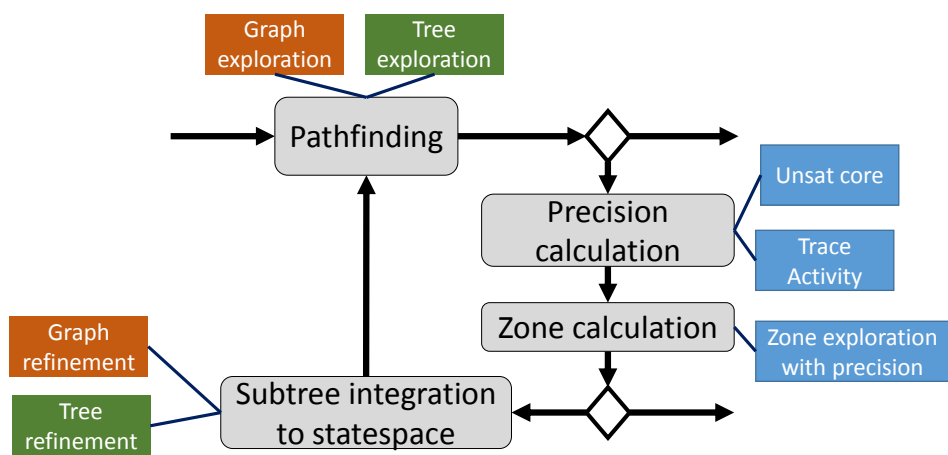
**Figure 3.5**    Statespace-based refinement

Chapter 4

# Implementation

## 4.1 Environment

### 4.1.1 Theta Model Checker

Theta Model Checkeris a framework developed by the Department of Measurement and Information Systemsthat offers model checking algorithms for various models, such as programs and statecharts. The models are described by domain specific languages, and translated to common formalisms, including the state transition system, the control flow automaton, and the timed automaton. Besides formalisms, abstract domains and frameworks for common model checking approaches are also implemented. Theta Model Checkeruses an SMT solver called Z3[1], that is able to recognize various first order theories, such as difference logic.

I have decided to extend Theta Model Checkerwith the presented configurable framework for model checking timed automata. The implemented framework relies on the model checker's extended timed automaton representation: the *Timed Control Flow Automaton*, Z3 interface, and a modified version (modifications described in chapter 3) of the zone implementation described in [1].

### 4.1.2 Achitecture

**TODO:** Ábra amin látszik ki-bemenet,Solver, stb

The basic architecture of the framework presented in chapter 3 is shown on **TODO:** ábra ref.

The input of the algorithm consists of an input of the problem (a timed automaton $\mathcal{A}$ and a location $l_{err} \in L(\mathcal{A})$), and a configuration of the algorithm: compatible implementations of the CEGAR phases, and their parameters (e.g. the bound of the bounded model checker).

---

[1]https://github.com/Z3Prover

The output of the algorithm can be an execution trace by which $l_{err}$ is reachable, *No* if $l_{err}$ is unreachable, or *Undecided*. The latter case can happen for two causes: either the computations on the discrete variables make the problem undecidable, or thr bounded model checker proved that $l_{err}$ is unreachable in the given number of steps.

## 4.2   Measurements

Measurements were performed on a personal computer with a core i5 processor. The program was operating on a maximum of **TODO:** (ecplipsenek adható max) memory.

### 4.2.1   Objectives

The goal of the measurements is to evaluate the designed algorithm's performance and scalability, and draw conclusions about what combination of algorithms are effective. Inputs were chosen from Uppaal's benchmark automata[2], but only the scalable ones. Uppaal supports extensions of the timed automaton formalism (synchronization channels) that are not implemented in Theta Model Checker. This was solved by generating the complete product automata instead of network automata.

### 4.2.2   Inputs

This section describes the input models used for measurements.

**Fischer's protocol**

**CSMA/CD protocol**

**Token Ring FDDI Protocol**

### 4.2.3   Results

**TODO:** Grafikonok + mit mértünk épp, mivel, mi lett az eredménye

### 4.2.4   Evaluation

**TODO:** Miérések eredményének összesítése, mit tudtunk meg ebből.

---

[2]https://www.it.uu.se/research/group/darts/uppaal/benchmarks/

# Chapter 5

# Related Work

**TODO:** Milyen más Timed CEGAR megközelítések vann, és ehhez képest a miénk miben más, és főleg mibven jobb.

Chapter 6

# Conclusions

**TODO:** Ha van valami nagyobb/meglepőbb eredmény, azt lehet hangsúlyozni.

## 6.1 Contribution

**TODO:** Szokásos pontokba szedett, részletes kontribúcióismertetés.

## 6.2 Future work

**TODO:** predikátum interpolánssal + egyebek Pl. paraméteres, vagy Ákossal összedolgozós, stb.

# References

[1]  Johan Bengtsson and Wang Yi. "Timed Automata: Semantics, Algorithms and Tools". In: *Lectures on Concurrency and Petri Nets*. Vol. 3098. LNCS. Springer Berlin Heidelberg, 2004, pp. 87–124.

[2]  Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007, pp. I–XV, 1–366. URL: http://dx.doi.org/10.1007/978-3-540-74113-8.

[3]  Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement for symbolic model checking". In: *Journal of the ACM (JACM)* 50.5 (2003), pp. 752–794.

[4]  C. Daws and S. Yovine. "Reducing the number of clock variables of timed automata". In: *Proceedings of the 17th IEEE Real-Time Systems Symposium (RSS '96*. IEEE, 1996, pp. 73–81. ISBN: 0-8186-7689-2.