Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

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. Differenece 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 $\bot$ (flase), and propositional variables $p, q, \cdots$ with the use of logical connectives. A formula ϕ can be an atom (a truth symbol or a variable) or can be constructed from other logical formulae with the following connectives:

- negation: $\neg\phi_1$ is evaluated true iff ϕ_1 is evaluated false (formal equivalent of 'not'),
- conjunction: $\phi_1 \wedge \phi_2$ is evaluated true iff both ϕ_1 and ϕ_2 is evaluated true (formal equivalent of 'and'),
- disjunction: $\phi_1 \vee \phi_2$ is evaluated true iff at least one of ϕ_1 and ϕ_2 is evaluated true (formal equivalent of 'or'),
- implication: $\phi_1 \rightarrow \phi_2$ is evaluated true iff ϕ_1 is evaluated false or both ϕ_1 and ϕ_2 is evaluated true (formal equivalent of 'if ... then'),
- equivalence: $\phi_1 \leftrightarrow \phi_2$ is evaluated true iff both ϕ_1 and ϕ_2 is evaluated true or both ϕ_1 and ϕ_2 is evaluated false (formal equivalent of 'if and only if').

Note, that 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 ϕ.

Output : *Yes* if ϕ is satisfiable (i.e. it is possible to ground the variables appearing in ϕ to truth symbols so that ϕ 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. Formuale 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 solveable for most practical problems. Formalization of structures is what bridges the gap.

Definition 2.1 A *first order theory* $\mathcal{T}$ is a pair $(\Sigma, \mathcal{A})$ where

- Σ 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 Σ 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 Σ-formula ϕ.

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

In many practical theories, SMT becomes solveable.

2.1.3 Difference logic

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 Z and n is a constant. A difference logic formula ϕ 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 adress the algorithms for deciding satisfiability (if it is decidable). For more information on satosfiability, the reader is referred to **TODO:** hivatkozás.

2.2 Formal verification

Formal verification is the act of proving the correctness or incorrectness of a system in a mathematically precise way. In order for that to happen, the system has to be represented by a formal model, and the requirements of the system has to be formally defined as properties of the given model. Verification can be performed by proving that the system (repreented by the formal model) satisfies that property. The property may be a structural criterion or an expectiation about the behaviour of the system.

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. *Structural models* are useed to represent structural properties, while *behavioral models* reresent behavioral properties. This paper focuses on the latter.

Finite automata

The finite automaton (plural: automata) is one of the most common formalisms for modeling behavior. This way the system's behavoiour is modelled by a finite set of possible states the system can step in, and a set of steps defining what states the system can step when it is about to change it's state.

Definition 2.2 Formally a *finite automaton*, or state machine $\mathcal{FA}$ is a touple $\langle S, s_0, T \rangle$ 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.

$\mathcal{FA}$ can be represented as a directed graph $G_{\mathcal{FA}}$ where $V(G_{\mathcal{FA}}) = S$ and $E(G_{\mathcal{FA}}) = T$.

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. This formalism is easy to use and veryfy, but its expressive power is not necessary – i.e. many types of behavoiurs 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.

A finite automaton extended by variables could be (briefly) defined as a touple $\langle L, l_0, E, I \rangle$ where

- L is the set of control locations,
- $l_0 \in L$ is the initial location,
- $E \subseteq L \times \mathcal{B} \times \mathcal{U} \times L$ is the set of edges (where $\mathcal{B}$ can be briefly described as the set of expressions constructed from the variables, operators and brackets that can be evaluated to true or false, and $\mathcal{U}$ can briefly defined as a set of functions assigning a new values to some variables, that can also be described by an expression constructed from variables, operators and brackets) and
- $I : L \to \mathcal{B}$ assigns invariants to locations.

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 aswell, 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 assignment 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.

Initially the system is in l_0. Variables do not have initial values, thus they only get values when until one is assigned to them by some transitions. 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 doesn't assign a new one).

TODO: példa

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 states and the desired outcome of the verification is that it is unreachable.

The problem can be defined as follows.

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

: *Yes* if the S can reach s_{err}, *No* if not.

It is also useful, if the answer is *Yes*, to provide a *counterexample*: an execution trace $\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} s_{err}$, setting the systems's state to s_{err}.

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.

: A finite automaton $\mathcal{FA}$ and a state s_{err}.

: A sequence of states and outgoing transitions $\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} s_{err}$ if the S can reach s_{err} , *No* if not.

This problem can be solved by any pathfinding algorithm.

In case of an finite automata with variables the problem can be interpreted as

: A finite automaton (extended with variables) $\langle L, l_0, E, I \rangle$ and a control location l_{err}.

: A sequence of locations and enabled transitions $\sigma = l_0 \xrightarrow{e_0} l_1 \xrightarrow{e_1} \cdots \xrightarrow{e_n} l_{err}$ 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 a pronlem like this.

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

Approaches

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 2.3 The statespace of a system is the set of states that are reachable from the initial state by 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 chosing 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 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, however, sometimes, when small counterexamples are expected, it is not necessary.

Consider the *bounded reachability problem*.

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

: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.

–>boundedness ->Abstraction –> (FA pl- változós) –>CEGAR

The CEGAR approach introduced in [2] makes abstraction refinement a key part of model checking. The idea is illustrated on Figure 2.1.

First, an abstract system is constructed. The key idea behind abstraction is that the state space of the abstract system overapproximates that of the original one. Model checking is performed on this abstract model. If the target state is unreachable in

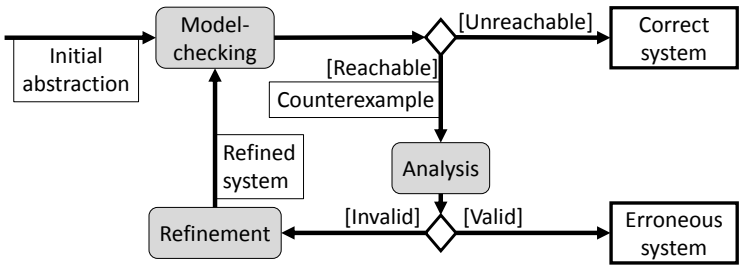

Figure 2.1 Counterexample-guided abstraction refinement

the abstract model, it is unreachable in the original model as well. Otherwise the model checker produces a counterexample – a run where the system reaches the target state. In our case the counterexample is a sequence of transitions – i.e., a trace. Overapproximation brings such behaviors 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. 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 given leading to the erroneous state

TODO: –>SAT

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

TODO: példák!

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 betrween two modifications. However, this is not the case for clock variables (clocks, for short). Even when a system stays in one state, the value of clocks are continuously and steadily increasing. Naturally, their values can be modified, but the only allowed operation on clock variables is *reset*. Reseting a clock means assigning its value to a specific integer (often, that integer can only be 0). It's an instantaneous operation, after which the value of the clock will continue to increase.

TODO: clock definícióját kiemelni

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

Definition 2.4 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 2.5 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. Clock constraints are used in guards and invariants of timed automata to control the behaviour by only allowing certain operations if the current valuation satisfies the consraints. **TODO:** ezt nem kell ennyire szájbarágósan, ha korábban már definiálva van a guard, inv, etc.

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

Definition 2.6 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. Invariants can be used to ensure the progress of time in the model. [1]

The automaton's edges, that are defined by the source location, the guard (represented by a clock constraint), the set of clocks to reset **TODO:** ez az update TA-ra, and the target location. Graphically a timed automaton can be represented as a labeled graph where the vertices are the locations labelled with their corresponding invariants. **TODO:** referálnmi az FA-ra

TODO: példa

Definition 2.7 A *state* of $\mathcal{A}$ is a pair $\langle l, v \rangle$ where $l \in L$ is a location and $v : \mathcal{C} \to R$ 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 2.8 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 2.9 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 $I(l)$.

There are many extensions of the timed automata formalism (e.g. this definition only allows to reset clocks to 0, however, resets to greater integers will appear later in this paper **TODO:** biztos?). 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. Bool, integer, rational, or even self-described typed variables prove useful, but may result in a formalism with bigger expressive power that that of the conventional timed automaton. This becomes important when one wants to analyze a system.

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.

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.

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

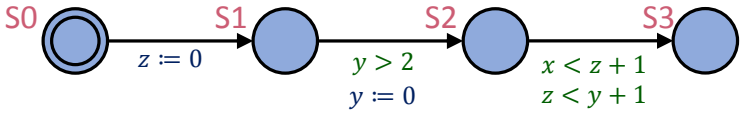

Figure 2.2 Timed automaton example

Definition 2.10 A *zone* z is a set of nonnegative clock valuations satisfying a set of clock constraints.

Definition 2.11 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.

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 statspace. 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 $\langle l, z'\rangle$.

For ease of understanding the algorithm is demonstrated on the automaton in Figure 2.2. The inital 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 $z \leq x \wedge x = y \wedge y > 2$). Th 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 $\langle S2, z \leq x \wedge y \leq z \wedge x - y > 2 \rangle$.

There outgoing transition from location $S2$ has a guard $x < z + 1 \wedge z < y + 1$ 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.

Unfortunately, it is possible that the graph described by the previous algorithm becomes infinite. Consider for example the automaton from [1] in Figure 2.3. 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.

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

From this node, both transitions are enabled. The loop transition increases the difference between x and y yielding the new node $\langle loop, x \leq 10 \wedge y \leq 30 \wedge y - x = 20 \rangle$, 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 2.3.

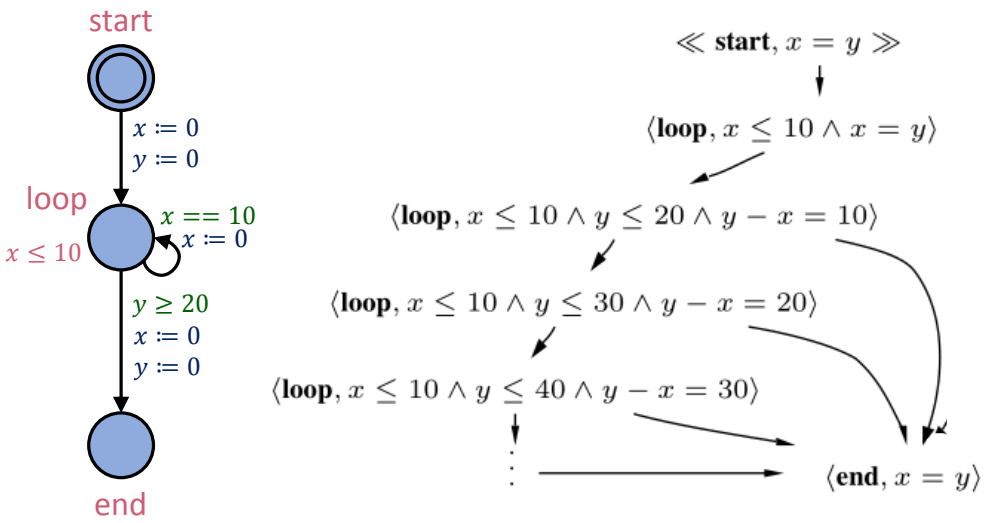

Figure 2.3 Timed automaton with infinite zone graph

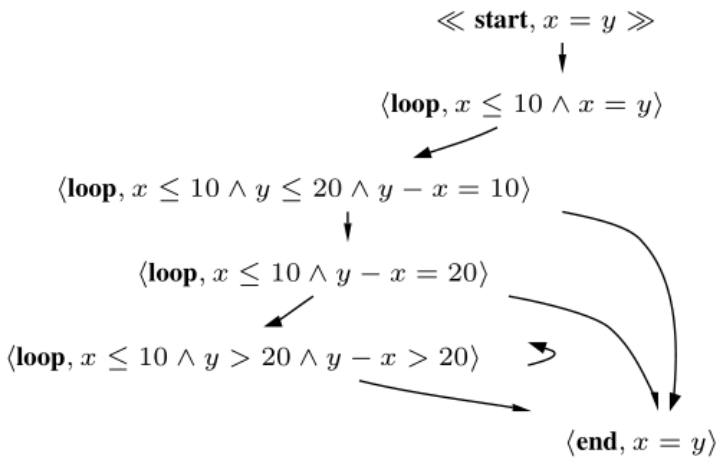

Figure 2.4 Finite zone graph

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).

In the automaton depicted in Figure 2.3, $k(y) = 20$ (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. Implicitely $y \leq 40$ 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 2.4.

Using normalization the zone graph is finite, but unreachable states may appear

in it. If the automaton doesn't have any guard or invarint 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.

To demonstrate the incorrectness of the algorithm, consider again the automaton in Figure 2.2. Recall that the reachabble 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.

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).

To demonstrate the effects of split, let us construct the zone graph of the automaton of Figure 2.2. 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: $\langle S1, x = y \wedge x - z < 1\rangle$ and $\langle S1, x = y \wedge x - z \geq 1\rangle$.

From $\langle S1, x = y \wedge x - z < 1\rangle$, $\langle S2, x - z \leq 1 \wedge z - y \leq 1\rangle$ is reachable, where the transition to location $S3$ is not enabled because of the guard $x - z < 1$.

From $\langle S1, x = y \wedge x - z \geq 1\rangle$ the resulting zone after firing the transition is split along the constraint $z - y < 1$, resulting in nodes $\langle S2, x - z \geq 1 \wedge z - y < 1\rangle$, and $\langle S2, x - z \geq 1 \wedge z - y \geq 1\rangle$. 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 a conjunctive formula of atomic constraints of the form $x - y < n$ or $x - y \leq n$ where $x, y \in \mathbb{C}$ and $n \in Z$.

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$x \leq n$, $x \in \mathbb{C}n \in Z$ a new vareiable has to be introduced.

Definition 2.12 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 $\mathbf{0}\ x \sim n$, $\sim \in \{\leq, <, =, >, \geq\}$ can be transformed into $x - \mathbf{0} \sim n$, and all clock constraints can be transformed into the desired form.

Definition 2.13 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.

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

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) wherem m is a difference constraint tells if D contains any states satisfying m without affecting D

up(D) calcualtes $D^{\uparrow}$

consistent(D) is used to decide if the zone represented by D contains any states

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,n) where $c \in \mathcal{C}$, $n \in Z$ resets c to n

norm(D, k) where $k : \mathcal{C} \to Z$ 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.

TODO: egy-két mondat arról, hogy mit hogyan használunk az algoritmusban

Implementations (pseudocodes) of these operations are provided in [1]. Termination of the algorithm is also proven, but it's complexity is exponantial 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 [3] 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.

TODO: példa

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 $c \in clk(g)$.

Activitiy is calculated by an iterative algorithm starting from $Act_0(l) = clk(l)$ for each $l \in L$. In the $i^{th} iteration \mathrm{Act}_i(l)$ is derived by edtending $Act_{i-1}(l)$ 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 $l \in L$.

Complexity

As it was mentioned, reachability for timed automata without discrete variables is decidable (but it it exponential). It was also mentianed before that reachability for finite autamata extended by (discerete) 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. **TODO:** referálni az FA-s részre és elmagyarázni, esetleg példával

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 discerete 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 chosing the most effective parts of the original algorithms can result in an even more effective algorithm then the original ones. **TODO:** diszkrét változókat megemlíteni

The architecture of the framework is illustrated on **TODO:** ábra: (két résszel az automatáshoz és az állapottereshez) amin látszanak hogy pontosan mik lesznek a dobozok (milyen interfészek) és mi megy köztük a nyilakon, stb. As one can see, there are two different realization of the CEGAR-loop. The reason for this is that not all implementations of the CEGAR-phases can be used interchangably, since there are two distict ways CEGAR-loop can be applied to timed automata. The key difference is the basis of the refinement. While the first approach **TODO:** ábra a) részét referálni starts from a pure automaton (without any clock variables) and extends the current automaton with some clocks in each iteration **TODO:** háttérismereteknél referálni - and thus refines the *automaton*, the other **TODO:** ábra b) részét referálni is based on the refinement of the *statespace* itself. Because of this, only algorithms of the same approach can be combined.

3.1.1 Automaton-based refinement

TODO: Fig ... depicts the architecture... The initial abstraction is a finite automaton that is derived from the original timed automaton by removing all clock variables and clock constraints.

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 is quite a complex task and therefore there aren't many implementations of it.

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 performed various ways. In this framework, the main idea is to explore the statespce without considering clock variables (and in some cases discerete 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. **TODO:** Fig ... depicts the architecture...

Because of the different approaches of abstraction, consructing the initial abstraction is not as straightforward as it was in the automaton-refinement phase. 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 a pathfinding algorithm.

The task of the analysis phase is to decide which of the clock variables to include in the zones on the trace so that it becomes possible to find out whether or not the counterexample is feasible. The result of the analysis should be a function $P : V(G) \to 2^{\mathcal{C}}$

assigning a set of clocks to the nodes of the current abstraction of zone graph. This set of clocks can be callled the *precision* of the zone.

The task of the refinement phase is to calculate the zones on the trace (up to the given precision) and find out if it was feasible or not. This can be performed by the steps of the algoorithm presented in **TODO:** utalás háttérismeretek megfelelő részére with some modifications that help with handling the changes of precision along the 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 current abstraction of the zone graph must be modified accordingly.

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 interchangably, that are defined for the same representation.

3.2 Modules

This section describes the implementations of the previously defined interfaces - grouped by the base of refinement.

3.2.1 Implementations for automaton-based refinement

First, model checkers are presented that can be used for the model checking phase of CEGAR algorithms with automaton-based refinement. Secondly, an algorithm is defined for calculating the set of clock variables to refine the automaton, and finally, the general algorithm is described for performing the refinement.

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 **TODO:** ref explains).

Satisfiability-based model checker

Satisfiability-based model checking was introduced in **TODO:** background ref. The idea can be directly applied to timed automata – the only necessary change is to define 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 σ 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 σ 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 **TODO:** 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 **TODO:** solver constraints? the same way as guards.

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

TODO: példa

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 σ is found from l_0 to l_{err}, it can be checked, and if the derived formula is satisfiable, σ 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:** háttérismerekhez ez is.

TODO: A pseudocode is provided to demonstrate implementability.

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 *unsat* - 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.

TODO: példa

TODO: pszeudokód?

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.

TODO: a végére ábra az elkészült dobozokról.

3.2.2 Implementations for statespace-based refinement

First, two possible representations of the abstract zone graph are presented. After that implementations of the CEGAR phases are presented, mentioning the statespace representation-dependent behaviours of phases model checking and refinement.

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 the zones will always be denoted: 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.

Because of this the initial abstraction can be constructed by assigninga 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 $\langle l', z_\emptyset \rangle$.

TODO: példa

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.

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 sttespace exploration.

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.

TODO: példát folytatni eszerint

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 introduce 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.

TODO: példa folytatása eszerint

TODO: pszeudokód

Trace Activity

The task of the analysis phase is to determine the precision of each zones on a given counterexample. The abstracion *activity* as described in **TODO:** háttérism referálni 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 formulae $Act_\sigma(n_i) = (Act_\sigma(n_{i+1}) \setminus r_i) \cup clk(g_i) \cup clk(I(l_i))$.

TODO: példa

Unsat core-based precision

Unsat core can also be used to determine the necessary precision of a given counterexample. First, the SAT formula described in part 3.2.1 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 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.

TODO: példa

Statespace refinement

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.

Let us 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.

TODO: példa

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.

TODO: pszeudokód az eddigiekről

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.

Discree 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.

TODO: példa,pszeudokód?

Incoming edges that are not on the trace are also important in case of tree representatio, 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.

TODO: példa, pszeudokód?

3.3 Result

TODO: Kis szöveg meg sok sok ábra az elkészült algoritmusokról, a ko9mbinálhatóságról, valamint a keretrendszer kiterjesztési lehetőségeiről

Chapter 4

Implementation

4.1 Environment

TODO: TTMC bemutatása,stb

4.2 Measurements

4.2.1 Objectives

TODO: Célok ismertetése, mérések bemutatása. Mit akarunk mérni, mivel fogjuk összehasonlítani, milyen bemeneteken, és miért.

4.2.2 Inputs

TODO: Uppaal inputok, stb.

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.

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] 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.

[3] 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.