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

TODO: Ákos dipterv

2.1 Mathematical logic

2.1.1 Zeroth order logic

SAT

2.1.2 First order logic

SMT

2.2 Formal verification

TODO: Importance, etc.

2.2.1 Timed automata

TODO: Modeling formalisms, timed systems, etc.

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 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.

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

Definition 2.1 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 2.2 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.

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

Definition 2.3 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]

Graphically a timed automaton can be represented as a labeled graph where the vertices are the locations labelled with their corresponding invariants, and the edges are the automaton's edges, that are defined by the source location, the guard (represented by a clock constraint), the set of clocks to reset, and the target location.

TODO: példa

A state of $\mathcal{A}$ is a pair $\langle l, v \rangle$ where $l \in L$ is a location and v 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. 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 $I(l')$.

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 variations of timed automata (e.g. this definition only allows to reset clocks to 0, however, resets to greater integers will appear later in this paper). 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.

2.2.2 Reachability

TODO: Importance, basic algorithms (statespace exploration, SAT based solution, bounded stuff + examples), TA reachability + examples

Timed automaton reachability

TODO: complexity, complextity w/ disc vars, encoding disc vars in locations, etc

2.2.3 CEGAR

Abstraction

Idea, usefulness, Times automata - zones, variables, activity, etc. +precisiont is definiální

CEGAR-loop

Idea, Cegar-loop, basic cegar ideas (variable-based, statespace refinement, etc.)

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 2.2.2 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 2.2.2, 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.