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

Rebeka Farkas

CEGAR-based analysis of timed systems

MSc Thesis

Supervisor:

András Vörös

Budapest, 2016

Contents

Kivonat A valósidejű biztonságkritikus rendszereken alkalmazott formális verifikáció képes felfedezni tervezési hibákat a fejlesztés különböző fázisaiban. Azonban a formális módszerek nagy számításigénye gyakran gátat szab a sikeres verifikációnak.

Az absztrakció módszerét gyakran alkalmazzák egyszerű, hatékonyan verifikálható modellek megalkotására, az ellenpélda vezérelt absztrakciófinomítás (counterexample-guided abstraction refinement, CEGAR) iteratív módszere segítségével pedig megválasztó a megfelelő szintű absztrakció.

A hatékony verifikációhoz fontos a megfelelő modellezőeszköz megválasztása. 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.

Az irodalomban sokféle algoritmus található időzített automaták verifikálására, melyek közül saját algoritmusom alapjául egy széles körben elterjedt, hatékony modellellenőrző, az Uppaal algoritmusa szolgál. Ennek különlegessége, hogy egy hatékony absztrakciót, úgynevezett zónaákat használ a folytonos, így végtelen állapottér reprezentálására.

Dolgozatomban bemutatok egy új CEGAR-alapú megközelítést, amely lehetővé teszi időzített automatákkal leírt valósidejű rendszerek formális verifikációját. Az algoritmus (beleértve az Uppaal modellellenőrző algoritmusát) az implementálhatóság biztosítása érdekében részletesen bemutatásra kerül, az érthetőség megkönnyítése érdekében pedig egy példán is illusztrálva van. A dolgozat tárgyalja az algoritmus alkalmazhatóságát is, így bemutatja az előnyeit a korábbi, hasonló megoldásokhoz képest, valamint bizonyítást ad az algoritmus helyességére és terminálódására.

TODO : Befejezni

Kulcsszavak időzített automaták, elérhetőségi analízis, CEGAR

Abstract The verification of safety-critical real-time systems can find design problems at various phases of the development or prove the correctness. However, the computationally intensive nature of formal methods often prevents the successful verification. Abstraction is a widely used technique to construct simple models that are easy to verify, while counterexample guided abstraction refinement (CEGAR) is an algorithm to find the proper abstraction iteratively. In this work we extend the CEGAR framework with a new refinement strategy yielding better approximations of the system. A prototype implementation is provided to prove the applicability of our approach.

TODO: Lefordítani a magyart

Keywords timed automata, reachability analysis, CEGAR

Hallgatói nyilatkozat

Alulírott **Farkas Rebeka** szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt : Budapest, 2016. július 27.

....
Farkas Rebeka

Chapter 1

Introduction

It is important to be able to model and verify timed behavior of real-time safety-critical systems. One of the most common timed formalisms is the timed automaton that extends the finite automaton formalism with real-valued variables – called clock variables – representing the elapse of time.

A timed automaton can represent two aspects of the behavior. The discrete behavior is represented by locations and discrete variables with finite sets of possible values. The time-dependent behavior is represented by the clock variables, with a continuous domain.

A timed automaton can take two kinds of steps, called transitions: discrete and timed. A discrete transition changes the automaton's current location and the values of the discrete variables. In addition, it can also reset clock variables, which means it can set their value to 0. Time transitions represent the elapse of time by increasing the value of each clock variable by the same amount. They can not modify the values of discrete variables. Transitions can be restricted by guards and invariants.

In case of real-time safety-critical systems, correctness is critical, thus formal analysis by applying model checking techniques is desirable. The goal of model checking is to prove that the system represented by the model satisfies a certain property, described by some kind of logical formula. My research is limited to reachability analysis where the verification examines if a given set of (error) states is reachable in the model. Reachability criterion defines the states of interest.

Many algorithms are known for model checking timed systems, the one which defines an efficient abstract domain to handle timed behaviors is presented in [1]. The abstract domain is called *zone*, and it represents a set of reachable valuations of the clock variables. The reachability problem is decided by traversing the so-called *zone graph* which is a finite representation (abstraction) of the continuous state space.

Model checking faces the so-called state space explosion problem – that is, the state space to be traversed can be exponential or even larger compared to the size of the system. It is especially true for timed systems: complex timing relations can necessiate a huge number of zones to represent the timed behaviors. A possible solution is to use abstraction: a less detailed system description is desired which can hide unimportant parts of the behaviors providing less complex state space representations.

The idea of counterexample-guided abstraction refinement (CEGAR) [2] is to apply model checking to this simpler system, and then examine the results on the original one. If the analysis shows that the results are not applicable to the original system, some of the hidden parts have to be re-introduced to the representation of the system – i.e., the abstract system has to be refined. This technique has been successfully applied to verify many different formalisms.

Several approaches have been proposed applying CEGAR on timed automata. In [6] the abstraction is applied on the locations of the automaton. In [7] the abstraction of a timed automaton is an untimed automaton. In [4, 5], and [8] abstraction is applied on the variables of the automaton.

My goal is to develop an efficient model checking algorithm applying the CEGAR approach to timed systems. The above-mentioned algorithms modified the timed automaton itself: my new algorithm focuses on the direct manipulation of the reachability graph, represented as a zone graph, which can yield the potential to gain finer abstractions.

TODO: befejezni

Chapter 2

Background

This chapter defines the important aspects of timed automata and briefly explains the reachability algorithm presented in [1] and demonstrates it on some examples (based on examples, also from [1]). CEGAR is also explained at a high level.

2.1 Basic Definitions

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. For bravity, the term *clock* is used as a synonym of clock variable.

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.

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, $I : L \rightarrow \mathcal{B}(\mathcal{C})$ assigns invariants to locations. Invariants can be used to ensure the progress of time in the model.

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* 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 $I(l)$.

2.2 Reachability Analysis

A *zone* is a set of nonnegative clock valuations satisfying a set of clock constraints. The set of all valuations reachable from a zone z by time transitions is denoted by $z^{\uparrow}$.

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. Therefore, a node denotes a set of states. Edges between nodes denote transitions.

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 $z^{\uparrow}$. 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 $\langle l, z'\rangle$, $\langle l, z\rangle$ can be removed, and the incoming edge should be redirected to $\langle l, z'\rangle$.

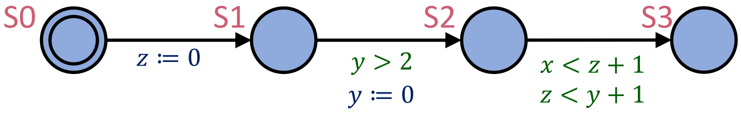

Figure 2.1 Timed automaton example

Example 2.1 For ease of understanding the algorithm is demonstrated on the automaton in Figure 2.1. 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 we can determine $x < y+2$ contradicting the atomic constraint $x - y > 2$ in the reachable zone of location $S2$. Thus the transition is never enabled, and location $S3$ is unreachable.

The resulting zone graph is the following.

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

Unfortunately, it is possible that the graph described by the previous algorithm becomes infinite. Consider for example the automaton from [1] in Figure 2.2.

Example 2.2 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 $\langle loop, x \leq 10 \wedge y \leq 30 \wedge y - x = 20 \rangle$,

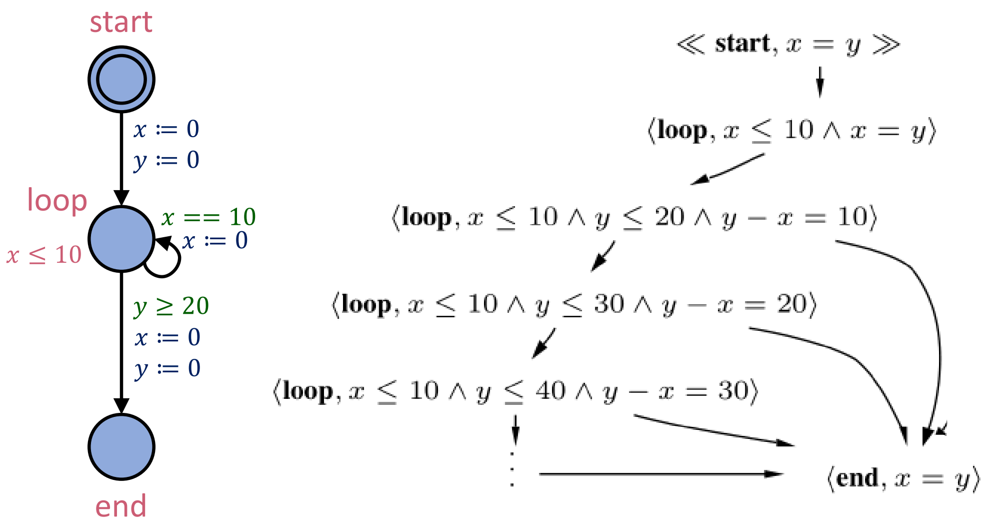
start
$x := 0$
$y := 0$
loop
$x \leq 10$
$x == 10$
$x := 0$
$y \geq 20$
$x := 0$
$y := 0$
end
$\ll$ **start**, $x = y \gg$
$\langle$**loop**, $x \leq 10 \wedge x = y\rangle$
$\langle$**loop**, $x \leq 10 \wedge y \leq 20 \wedge y - x = 10\rangle$
$\langle$**loop**, $x \leq 10 \wedge y \leq 30 \wedge y - x = 20\rangle$
$\langle$**loop**, $x \leq 10 \wedge y \leq 40 \wedge y - x = 30\rangle$
$\langle$**end**, $x = y\rangle$

Figure 2.2 Timed automaton with infinite zone graph

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 2.2.

In order for the zone graph to be finite, a concept called *normalization* is introduced in [1]. Let $k(c)$ denote the greatest value to which clock c is compared in the automaton. For any valuation v such that $v(c) > k(c)$ for some c, each constraint in the form $c > n$ is satisfied, and each constraint in the form $c = n$ or $c < n$ is unsatisfied, thus the interval $(k(c), \infty)$ can be used as one abstract value for c.

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

Example 2.3 In the automaton depicted in Figure 2.2, $k(y) = 20$ (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 2.3.

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.

Example 2.4 To demonstrate the incorrectness of the algorithm, consider again the automaton in Figure 2.1. 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

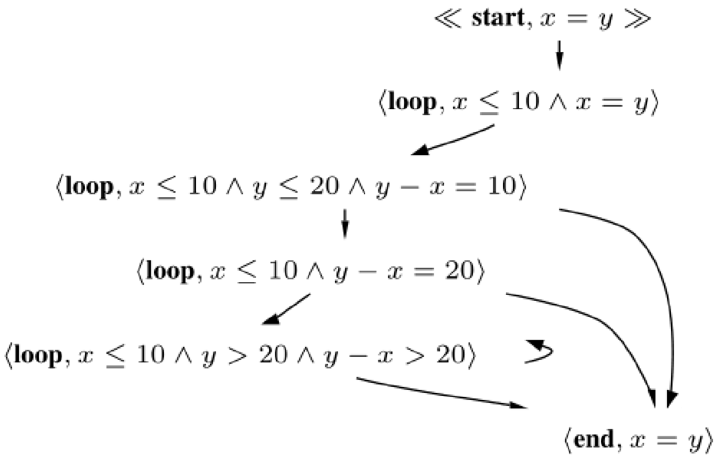
≪ start, $x = y$ ≫
⟨loop, $x \leq 10 \wedge x = y$⟩
⟨loop, $x \leq 10 \wedge y \leq 20 \wedge y - x = 10$⟩
⟨loop, $x \leq 10 \wedge y - x = 20$⟩
⟨loop, $x \leq 10 \wedge y > 20 \wedge y - x > 20$⟩
⟨end, $x = y$⟩

Figure 2.3 Finite zone graph

> $\langle S2, z \leq y \leq z \leq y \wedge x - y > 1\rangle$, 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).

> **Example 2.5** To demonstrate the effects of split, let us construct the zone graph of the automaton of Figure 2.1. 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].

2.2.1 Implementation

TODO: DBM-ről, műveletekről, komplexitásról beszélni!!!

2.3 CEGAR

The CEGAR approach introduced in [2] makes abstraction refinement a key part of model checking. The idea is illustrated on Figure 2.4.

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 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.

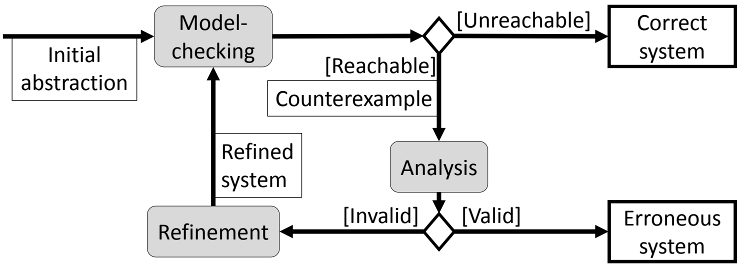

Figure 2.4 Counterexample-guided abstraction refinement

Chapter 3

Applying CEGAR to the Zone Graph

In this section an approach of applying CEGAR to the timed automaton is explained. Some details of implementation are also discussed.

3.1 Existing algorithms

As it was mentioned in Chapter 1, similar approaches have been presented in papers [4, 5] and [8]. The key idea behind these approaches is to apply abstraction by removing clock variables (and the constraints they appear in) from the automaton. The CEGAR loop is interpreted very similarly in these approaches.

Initial abstraction The location graph (i.e. the automaton without any clock variables) is used as an initial abstraction.

Model checking Model checking is applied on the current abstraction of the automaton using *Uppaal*'s algorithm as a module.

Analysis Uppaal is also used to simulate the counterexample on the original automaton.

Refinement Trivially, clock variables (and constraints they appear in) are added to the automaton, but the important part of the task is choosing which of the clocks to add. There are various approaches for that, including checking where the simulation stops with a synchronized linear automaton, using a SAT solver to analyze the trace expression, and a special algorithm developed for this purpose.

It is easy to see that these algorithms all rely on Uppaal's reachability algorithm (explained in Chapter 2), but all of them uses it as a 'black box' algorithm – they run it on different inputs, but they do not modify the algorithm itself. The disadvantage of these approaches, is that Uppaal has to run the same reachability analysis over and over on similar automata without storing any information from the previous calculations.

For example, for some traces even the initial abstraction is enough to see that the error location is unreachable that way. Since CEGAR-based algorithms focus on the traces where the error location is reachable, the model-checking algorithm will keep unnecessarily exploring the same traces (where the location is not reachable) – with growing complexity in each iteration as the automaton is refined. On the other hand – especially when using depth first search – the counterexamples may be similar: the first few locations and transitions of two consecutive counterexamples are usually the same. Considering this during the analysis of the counterexample could increase efficiency, since the simulation of the first few steps are already completed, it can continue from the first step that differs.

3.2 Introducing a new algorithm

TODO: El kéne nevezni ezt az algoritmust és akkor nem kellene mindenhol "az X. fejezetben bemutatt algoritmus"-ként hivatkozni rá.

My algorithm is explained in this section. To ease understanding it is also demonstrated on the automaton in Figure 2.2, with the error location being the location end.

3.2.1 Overview

Due to these disadvantages discussed above I have decided that my approach of applying CEGAR to the reachability analysis of timed automata will modify the reachability algorithm instead of using it as a black box module. My approach applies abstraction to the zone graph of the automaton, instead of the automaton itself. The reachability algorithm (which will now be a CEGAR-based algorithm) will refine the zone graph iteration by iteration until reachability can be decided. The CEGAR loop is interpreted the following way.

Initial abstraction The key problem about constructing the initial abstraction of the zone graph is that the zone graph is unknown so the abstraction has to be derived from the automaton itself. The idea is really simple: just like the other approaches I also use the location graph of the automaton as the initial abstraction, except in my algorithm it is considered to be the abstraction of the zone graph, not the automaton. To create an overapproximation of the zones, we simply consider every valuations to be reachable in all locations. The zone containing all valuation is denoted by z_∞.

Model checking Since the abstract zone graph is an abstraction of the reachability graph, model checking becomes a pathfinding problem in the current abstraction of the zone graph. The error location is either proven unreachable or a new trace (path in the graph) is found from the initial node to the target node.

Analysis This part is about finding out if the error location is really reachable on the trace found in the model-checking phase. The way to do that is by finding out how this path of the abstract zone graph would look like in the refined (real) zone graph. This can be achieved by using the reachability algorithm, but only for the given trace. As discussed in Section 2.2, because of the operation *split* sometimes the real zone graph can branch, which means that the result of the simulation may be a tree instead of a simple path. Nevertheless, it is still easy to decide whether the counterexample is valid: if the error location could be reached by the path (on any branch), then the counterexample is valid. Otherwise, simulation will stop somewhere, typically because one of the transitions (on each branch) is not enabled. In this case the counterexample is spurious.

Refinement In order to avoid the discussed disadvantages, this algorithm stores as many information of the analysis phase as it can - by replacing the counterexample trace in the abstraction of the zone graph with the calculated subgraph (tree) – thus refining the abstraction.

3.2.2 Details

It is very important to perform the presented operations correctly. This section explains the algorithm step by step, demonstrating it on the example automaton in Figure 2.2.

Constructing the initial abstraction is very straightforward: each node of the location graph are to be completed with the zone z_∞. After that, model checking is simply a pathfinding in the current abstraction of the zone graph.

> **Example 3.1** The initial abstraction of the example automaton's zone graph is depicted in Figure 3.1. The first counterexample is denoted with bold arrows.

Simulation of the counterexample is performed by constructing the relevant part of the real zone graph. In the first iteration each node on the path will contain z_∞. In this case, refinement starts from the node that belongs to the initial location and the refined zone is calculated as the initial zone of the zone graph.

The result of pathfinding in the graph in Figure 3.1 is denoted by bold arrows. In case of the later iterations the first few nodes of the trace will already be refined, so the refinement can start from the first abstract node. The reachable zone should be calculated from the last refined zone, considering the guards and the reset as when constructing the zone graph.

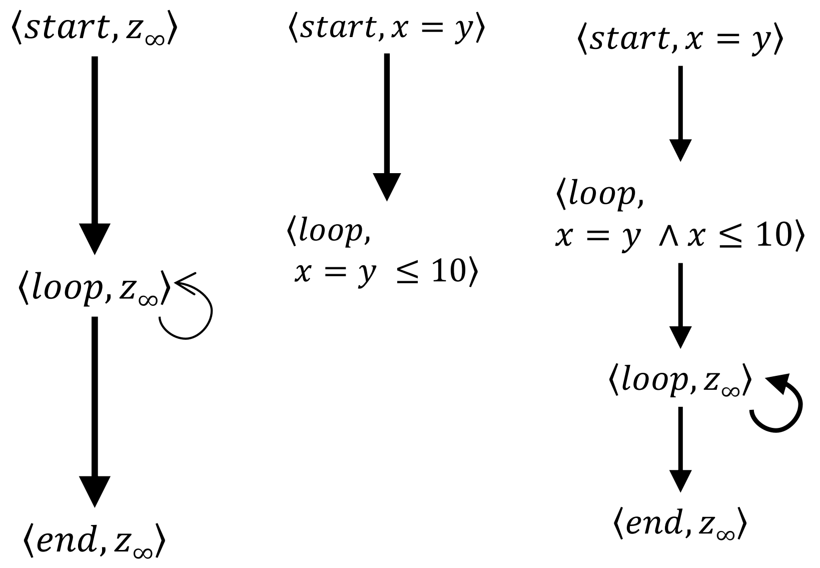

Figure 3.1 Initial abstraction

Figure 3.2 Result of analysis

Figure 3.3 After refinement

Figure 3.4 Timed CEGAR on an example

When the result of the refinement is more than one zone, the node on the path (and the edge pointing to it) is replicated, and one of the refined zones are assigned to each resulting node. The refinement can be continued from any of these nodes – the path branches. All of these branches should be analyzed (refined) one by one.

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.

> **Example 3.2** Consider the example. Since it is the first iteration, we start by constructing the initial node, $\langle start, x = y\rangle$. After that we calculate the next node on the trace $\langle loop, x = y \leq 10\rangle$. When constructing the zone graph, we continued with the transition represented by the loop-edge but this time we only have to explore the zone graph through the transitions in the counterexample. The next transition is the transition represented by the edge directed to node $\langle end, z_\infty\rangle$. This transition is not enabled in the previously calculated zone, which means the counterexample is spurious. The resulting subgraph of the zone graph is depicted in Figure 3.2.

The goal of refinement is to eliminate the spurious counterexample from the abstract representation. Refinement is applied by replacing the abstract counterexample with the subgraph of the real zone graph calculated in the *Analysis* phase. This operation has to be performed very carefully.

Consider e.g. that the node in the abstract graph that is about to get replaced by one (or more) nodes in the subgraph has other incoming edges than the one in the counterexample. Since it is unknown what states are reachable in the location by the other incoming edges, the node can't be removed. Except, the edge representing the transition in the counterexample has to be redirected to the node with the calculated zone (if there are multiple nodes, it has to be replicated).

Let us suppose that the graph is prepared to place the new node(s). To avoid wasting memory it is advised to use already refined zones in the graph. If the refined zone z of the node $\langle l, z \rangle$ is a subzone of a zone z' in a node $\langle l, z' \rangle$ (both nodes contain the same location l), then any state that is reachable from $\langle l, z \rangle$ is also reachable from $\langle l, z' \rangle$, thus there is no need to add $\langle l, z \rangle$ to the graph. Instead, any edge leading to $\langle l, z \rangle$ should be redirected to $\langle l, z' \rangle$. After that the replacement of the path can not continue from that $\langle l, z' \rangle$, since there might be more states reachable from $\langle l, z' \rangle$ than from $\langle l, z \rangle$. The remaining of the refined zones have to be recalculated and then the replacement can continue.

When replacing the node, the outgoing edges should also be considered. The calculated subtree of the real zone graph only contained the edges in the trace, but in the abstract zone graph there are other outgoing edges of the node. These edges are the outgoing edges of the original (abstract) node in the zone graph, and they have to be replicated, as outgoing edges from the added node(s). After this we can continue with the next node.

> **Example 3.3** In the example replacing the initial node can simply be performed by replacing the zone z_∞ with the zone $x = y$. On the other hand, replacing the node $loop$ has to be performed carefully. Since the loop-edge is an incoming edge of the node and is not part of the counterexample, the node can not be replaced. Instead, the incoming edge on the trace is redirected to the new node (in other words, the one to the original node is removed). Since a new node is added, we have to think about it's outgoing edges that are not on the trace – in this case the only one is the loop edge. For reasons explained before, the loop edge has to be redirected to the abstract version of the node. Thus, the refinement is finished, and now the counterexample is eliminated from the abstract zone graph. Th resulting graph is depicted in Figure 3.3.

3.3 Evaluation

It is important to prove that the algorithm is correct, and that it terminates. Both proof relies on that of the original algorithm.

The algorithm always terminates, because the abstraction of zone graph gets refined in every iteration. Even at worst case, the algorithm stops when the zone graph is completely refined.

The algorithm is correct, because the original algorithm is correct. If the algorithm finds a counterexample, it is a trace in the original algorithm. Otherwise, if there is a counterexample in the original algorithm, this algorithm finds it at some point.

Chapter 4

Improvements

TODO: Kéne egy jobb cím, hogy ez lehessen alcím

This chapter provides some improvement ideas of the algorithm defined in Chapter 3. First, some additional background knowledge is presented that provides the base of the improvement methods. Secondly, the improvements

4.1 Background

TODO: Szerintem ennek itt külön kell lennie, mivel a nagy background részben furán venné ki magát. ...Igazából így is fura. Ezt még ki kell találni.

4.1.1 Online pathfinding

4.1.2 Activity, Equality

In [3] two abstractions are proposed to reduce the number of clock variables in timed automata, or rather in the locations of timed automata. These abstractions are *exact* (they hide unimportant parts of the system without losing information about the desired property) and can be calculated by simple preprocessing algorithms. The key idea is to examine each of the locations and tell if some clock variable is unnecessary, either because their value is never checked or because they are represented by other clocks having the same value. After these local reductions the number of necessary clock variables of the automaton can be defined by the maximum of necessary clocks in the locations.

Activity

By definition a clock c is considered *active* in a certain location l ($c \in act(l)$) if their value can affect the (future) operation of the system. Practically a clock is active between the point of reset and a point of check (by a guard or a location invariant). Naturally the reset and check events don't always come alternately as there are many paths going though the same location, so calculating activity is not trivial, but it is indeed simple.

It is easy to see that $c \in act(l)$ for any $c \in clk(l)$ – that is, the set of clocks appearing in the invariant of the location or in any of the guards of the outgoing transitions. The algorithm calculates $clk(l)$ for each location l and then expands these sets iteratively by the following rule: for each edge $e(l, g, r, l')$ $act(l) := act(l) \cup (act(l') \setminus r)$.

TODO: valami szép példa

Equality

Two clocks c_1 and c_2 are considered *equal* in a location l if $v(c_1) = v(c_2)$ for any valuation v reachable in l. The algorithm for calculating $equ(l)$ calculates pairwise equality between clocks. It is also iterative, starting from $equ(l) = \mathcal{C} \times \mathcal{C}$ for each location l and narrowing iteratively considering preceding locations and incoming edges.

Activity and equality can be combined, allowing a high level of abstraction without loss of precision.

Renaming

TODO: kell ez a rész?

The paper also suggest renaming methods to avoid unnecessary clock variables. E.g. if the automaton operates on n clocks and applying activity reveals that at most $k < n$ clock variables are needed in each locations, a table of size $n \times |\mathcal{L}|$ can be constructed to show which of the k new clock variables represent a given clock at a given location.

4.1.3 Interpolants

4.2 Methods of improvement

4.2.1 Model checking using online pathfinding

TODO: Ide még az Online pathfinding felhasználása Counterexamplekhez

4.2.2 Precision

In [3] activity and equality were treated as if they were some sort of structural properties of the automaton. Obviously, the presented preprocessing methods can be applied before any reachability algorithm (for timed automata), including the one presented in Chapter 3, but – as this section will prove – the idea of activity and equality has more to offer. First, an obvious remark to anyone operating on the reachability graph with zones implemented as difference bound matrices: only the rows and columns of active clocks have to be included in the stored matrix (and they can even be sorted into equality groups), thus the nodes of the graph require less space. (This way the renaming process is not necessary.)

It may be even more useful in the algorithm presented in Chapter 3. The described algorithm refines the path completely. Applying the upper-mentioned idea reduces the number of clocks but still provides more information than necessary to decide if a counterexample is feasible.

TODO: példa, ahol a trace activity jóval kevesebb órát tartalmaz, mint a sima

Instead of refining every node completely the idea is to select the necessary clock variables in each node of the graph, and only refine to that precision. Interestingly, this can be obtained very simply, using the idea of activity.

Trace Activity

Trace activity is a function that assigns to the nodes of a trace (a counterexample of the reachability graph of a timed automaton) the set of clocks whose values are necessary to decide if it is a valid execution trace.

Let us describe a trace by

$$\sigma : n_0 \xrightarrow{e_1} n_1 \ldots \xrightarrow{e_k} n_k = n_{error}$$

where n_i-s represent nodes and e_i-s represent edges of the zone graph (and not the automaton).

Calculating trace activity starts from the last node n_{error} containing the location l_{error}. Obviously, non of the clocks are necessary in this node since we are only interested in whether or not the location is reachable. However, if there is an invariant of the location l_{error}, then it is important for the affected variables to be included in $Act_\sigma(n_{k-1})$ to decide if the last transition of σ, represented by edge e_k is enabled. This can either mean $Act_\sigma(n_{error}) = \emptyset$ and $clk(Inv(l_{error}))$ – that is, the clocks apperaing in the invariant of the error location – will have to be considered when calculating $Act_\sigma(n_{k-1})$ or simply $Act_\sigma(n_{error}) = clk(Inv(l_{error)})$. (However, it is quite unusual for error locations to have invariants.) For other nodes trace activity can be calculated by

$$Act_\sigma(n_i) := Act_\sigma(n_{i+1}) \setminus r_k \cup g_k \cup clk(Inv(l_i))$$

for all $0 \leq i \leq k-1$ where r_k, g_k and l_i appear in the edge $e(l_i, g_k, r_k, l_{i+1})$ (of the automaton) represented by e_k.

TODO: példa

Application

Calculating trace activity is simple, but raises questions about operating on zones with different precision. The algorithm explained in Chapter 2 and it's implementation described in [1] provide information on calculating the next zone, but do not tell what to do when variables appear and disappear between operations.

Creating zones

The first task is to define when (at which operations) does the precision change. As it was stated before activity changes with reset and check events - clocks can apperar when reset is applied, and disappear after guard conditions are checked. Clocks can also disappear *before* guard conditions are applied – the clocks that are active in the source node only because they appear in the invarint of the location. Of course, the simple solution is to only modify the set of clocks after checking guard conditions and during reset operations.

The next task is to define these operations considering that the matrix has to remain in a closed form. Fortunately, because of the nature of difference bound matrices, operations effecting only one clock variable only modify the row and column belonging to that clock while the other parts (submatrix) remains unchanged. This allows us to simply delete unnecessary rows and columns of the matrix after the guard constraints are applied, and faciliates introducing new variables: new rows and columns have to be added and the values of the matrix elements have to be set to the value *reset* would set them.

Operations

These operations were defined to allow adding or removing clocks to and from the base set of a zone while it was created, but sometimes it is necessary to change the base set of a zone after creation. Consider for example, that during some iteration of CEGAR a counterexample t is found with the first few nodes already refined to some precision (because they are also the first part of a previously examined counterexample t'). Since trace activity is determined by the subsequent nodes which obviusly differ in t and t' it is possible that the trace activity by t requires different precison than the existing one.

Let n be the first node of t where $Act_t(n) \neq Act_{t'}(n)$. The problem is with the clocks in $Act_t(n) \setminus Act_{t'}(n)$. If n is the initial node the clocks can be simply added by adding new rows and columns and making everything equal to everything. Otherwise **TODO:** valami jobb ötlet, mint újraszámolni az egészet.

Inclusion

TODO: Megoldani. Visszatérni a fás megoldáshoz?

4.2.3 ...

TODO: további ötletek

4.3 Implementation details

TODO: Ha van implementáció, akkor annak a bemutatása, de ha nincs, akkor is lehet mondani meglátásokat

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.

[4] Henning Dierks, Sebastian Kupferschmid, and Kim Guldstrand Larsen. “Automatic abstraction refinement for timed automata”. In: *Formal Modeling and Analysis of Timed Systems, FORMATS’07*. Vol. 4763. LNCS. Springer, 2007, pp. 114–129.

[5] Fei He, He Zhu, William N. N. Hung, Xiaoyu Song, and Ming Gu. “Compositional abstraction refinement for timed systems”. In: *Theoretical Aspects of Software Engineering*. IEEE Computer Society, 2010, pp. 168–176.

[6] Stephanie Kemper and André Platzer. “SAT-based abstraction refinement for real-time systems”. In: *Electronic Notes in Theoretical Computer Science* 182 (2007), pp. 107–122.

[7] Takeshi Nagaoka, Kozo Okano, and Shinji Kusumoto. “An abstraction refinement technique for timed automata based on counterexample-guided abstraction refinement loop”. In: *IEICE Transactions* 93-D.5 (2010), pp. 994–1005.

[8] Kozo Okano, Behzad Bordbar, and Takeshi Nagaoka. “Clock Number Reduction Abstraction on CEGAR Loop Approach to Timed Automaton”. In: *Second International Conference on Networking and Computing, ICNC 2011*. IEEE Computer Society, 2011, pp. 235–241.