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 **TODO :** Megírni

Kulcsszavak időzített automatás, elérhetőségi anlízis, 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 and easy to verify models, 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. május 21.

.....
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 statespace to be traversed can be exponential in 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 using abstraction: a less detailed system description is desired which can hide unimportant parts of the behaviors providing less complex system 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 [5] the abstraction is applied on the locations of the automaton. In [6] the abstraction of a timed automaton is an untimed automaton. In [3, 4], and [7] 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.

The paper is organized as follows...

TODO: befejezni

Chapter 2

Background

This section 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, $I : L \to \mathcal{B}(\mathcal{C})$ 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 $v(c)$ otherwise 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 transition. 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 $\langle l, z' \rangle$.

Example 2.1 For ease of understanding the algorithm is demonstrated on the automaton on Figure 2.2. The inital state is $\langle start, z_0 \rangle$ where z_0 is a zone containing only te initial valuation $v_0(x) = 0, v_0(y) = 0$. 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 two clocks will be incremented by the same value, x and y has the same value in each valuation contained by $z_0^{\uparrow}$. Since there is no invariant in

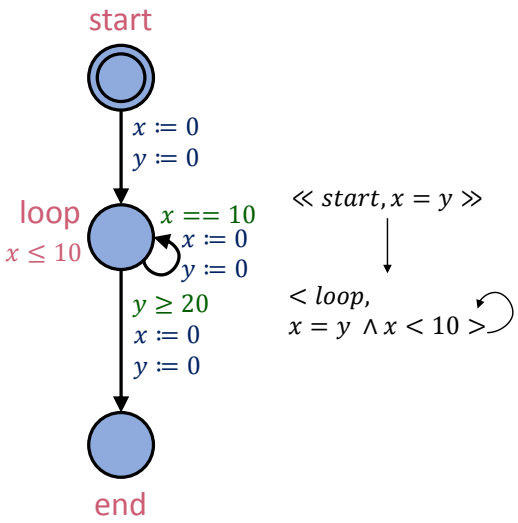

Figure 2.1 Timed automaton and it's zone graph

location $start$ x and y can take any positive value. Because of this z_0 can be defined by the constraint $x = y$ (that is, $x - y = 0$), and the initial node can be defined as $\langle start; x = y \rangle$.

There is only one outgoing transition from location $start$ and that resets x and y both, resulting in the same zone z_0 like before. The invariant $x \leq 10$ of location $loop$ is satisfied by z_0, but we have to consider it when calculating the valuations reachable by delay - only the states satisfying $x \leq 10$ of $z_0^{\uparrow}$ are reachable. The node of the graph can be defined as $\langle loop, x = y \wedge x < 10 \rangle$.

There are two outgoing transitions from this node. The loop transition resets the clocks resulting in z_0 again in node $loop$. From this we know that instead of creating a new node in the graph, this time we only have to create a loop-edge from the last node.

Since there is no state in $\langle loop, x = y \wedge x < 10 \rangle$ where $y \geq 20$ holds, the last transition will never be enabled, the zone graph is finished. The result can be seen on Figure 2.2.

Unfortunately the described graph can possibly be infinite. Consider for example the automaton from [1] on Figure 2.2. The only difference between this automaton and the one on figure 2.2, is that in this automaton the transition represented by the loop edge only resets x, but not y.

Example 2.2 Constructing the zone graph of this automaton starts similarly, with the nodes $\langle start, x = y \rangle$ and $\langle loop, x = y \wedge x < 10 \rangle$, where only the loop-transition

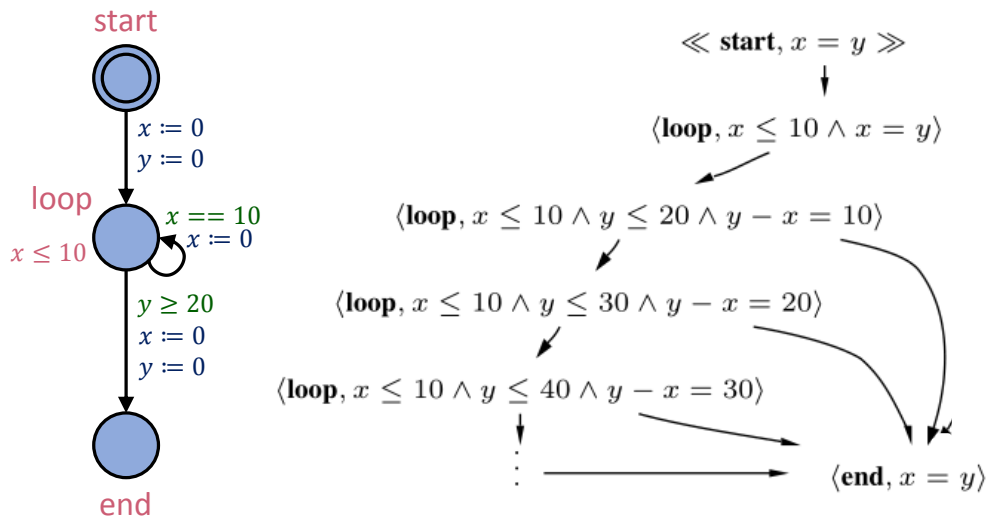

Figure 2.2 Timed automaton with infinite zone graph

is enabled.

This time the transition only resets x resulting in $\langle loop, x = 0 \wedge y = 10\rangle$. Because of the invariant, only 10 units of delay is enabled, resulting in the node $\langle loop, x \leq 10 \wedge y \leq 20 \wedge y - x = 10\rangle$.

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 on 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 valuations 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 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 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 on 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, depicted on 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.

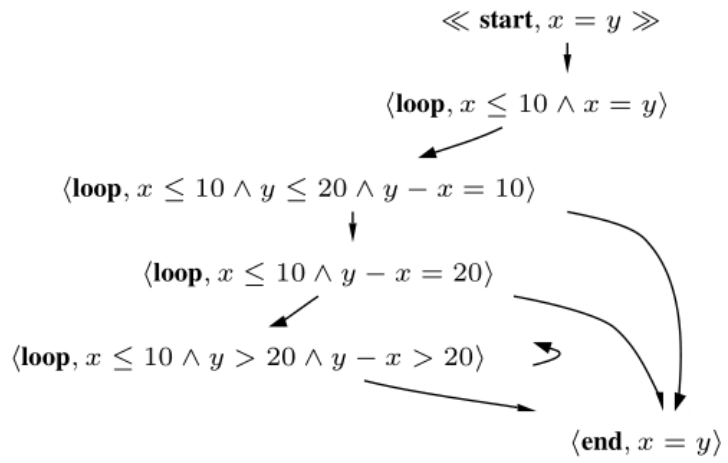

Figure 2.3 Finite zone graph

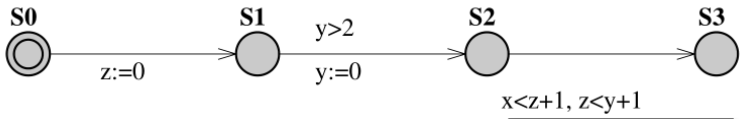

Figure 2.4 Timed automaton with incorrect graph after normalization

Example 2.4 To demonstrate the incorrectness of the algorithm, consider the automaton on Figure 2.4, with the error location $S3$. The reachabble states of the automaton (by calculations) are $\langle S0, x = y = z\rangle$, $\langle S1, z \leq x = y\rangle$ and $\langle S2, z \leq y \leq z \leq y \wedge x - y > 2\rangle$ - $S3$ is unreachable. Applying normalization results in the states $\langle S0, x = y = z\rangle$ (unchanged), $\langle S1, z \leq x = y\rangle$ (unchanged), $\langle S2, z \leq y \leq z \leq y \wedge x - y > 1\rangle$ (changed), making $S3$ seem 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 erpresenting 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 2.4. 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 enabledbecause 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.

2.3 CEGAR

The CEGAR approach introduced in [2] makes abstraction refinement a key part of model checking. The idea is illustrated on Fig. 2.5.

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 counterexample is found or when a feasible trace is given leading to the erroneous state.

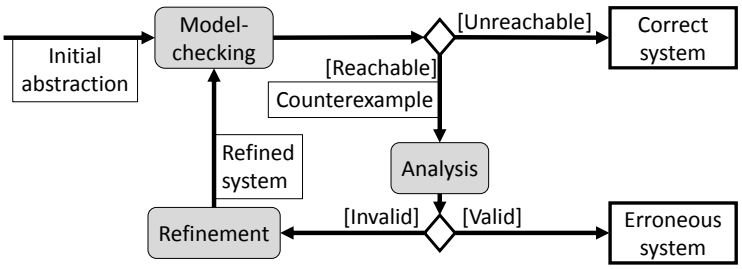

Figure 2.5 Counterexample guided abstraction refinement

Chapter 3

Applying CEGAR to the Zone Graph

In this section we explain our approach of applying CEGAR to the timed automaton. Some details of our implementation are also discussed.

3.1 Existing algorithms

TODO: Hasonló algoritmusok és hátrányaik bemutatása

3.2 Introducing a new algorithm

TODO: Szétszedni az ötletet, a megvalósítást, és a szemléltetést

Our algorithm is explained in this section. To ease presentation, we illustrate the algorithm on an example. The timed automaton is on Fig. **??** and the value of the parameter k is 2. The property to check is whether the automaton can reach the critical section.

3.2.1 Initial Abstraction

The first step of the CEGAR-approach is to construct an initial abstraction, which is an overapproximation of the system's state space. In our algorithm, the state space is represented by the zone graph. Instead of constructing the zone graph of the system an abstract simpler representation is constructed from the timed automaton.

Erroneous states are represented by (erroneous) locations, so we decided not to apply abstraction on them. However, the zones are overapproximated – the initial assumption is that every valuation is reachable at every location. This means that the initial abstraction of the zone graph will contain a node $\langle l, z_\infty \rangle$ for each location l, where z_∞ is the zone defined by the constraint set $\{c \geq 0 \mid c \in \mathcal{C}\}$.

Edges of the abstract zone graph can also be derived from the timed automaton itself. If there is no edge in the automaton leading from location l to l' there can not be a corresponding edge $\langle l, z\rangle \to \langle l', z'\rangle$ in the (concrete) zone graph regardless of z and z'. Thus, there should not be an edge from $\langle l, z_\infty\rangle$ to $\langle l, z'_\infty\rangle$ in the abstract zone graph either. All other edges are represented in the initial abstraction.

This results in a graph containing locations (extended with the zone z_∞) as nodes, and edges of the automaton (without guard and reset statements) – an untimed zone graph, derived completely from the automaton as the real zone graph is unknown. The initial abstraction derived from the example timed automaton can be seen on Fig. **??**.

This will be the model on which we apply model checking.

TODO: figure

3.2.2 Model Checking

During the reachability analysis only the counterexample traces will be refined in the zone graph. Thus, model checking becomes a pathfinding problem in the current abstraction of the zone graph in each iteration. Either we prove the target state to be unreachable or a new path is found from the initial node to the target node.

The result of pathfinding in the graph on Fig. **??** is denoted by bold arrows.

3.2.3 Counterexample Analysis and Refinement

Analyzing the counterexample in the original system and refining the abstract representation are two distinct steps of CEGAR, but in our approach they are performed together.

The goal of refinement is to eliminate the unreachable states from the abstract representation. Refinement is applied by replacing the abstract zones in the counterexample trace with refined zones containing only reachable states.

In the first iteration, no nodes of the abstract graph has ever been refined, so refinement starts from the node that belongs to the initial location where the refined zone is calculated from the initial valuation. 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 described in [1].

Of course, as discussed in Section 2.2 sometimes the result of the refinement is more than one zone. In this case the node in the graph (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.

It is also advised to reuse zones already refined. Suppose at one point of the algorithm the zone z_∞ of the node $\langle l, z_\infty\rangle$ is refined to z, and z is a subzone of a zone

z' in a node $\langle l, z' \rangle$ (both nodes contain the same location l). In this case any state that is reachable from $\langle l, z \rangle$ is also reachable from $\langle l, z' \rangle$, thus any edge leading to $\langle l, z \rangle$ is redirected to $\langle l, z' \rangle$, and $\langle l, z \rangle$ is removed. After that the analysis of the path can continue from that $\langle l, z' \rangle$.

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.

Let us consider the example. Refining the path on Fig. **??** is performed as follows. Refinements starts with the initial node $\langle S0, z_\infty \rangle$. First, we must consider the edge $\langle C1, z_\infty \rangle \to \langle S0, z_\infty \rangle$. The refinement will eliminate any state that is unreachable in the initial node of the zone graph, but there might be another node in the real zone graph with the location $S0$, so we duplicate the node before the refinement and the edge $\langle C1, z_\infty \rangle \to \langle S0, z_\infty \rangle$ will point to the duplicate. The value of x_1 is 0 in the initial state. Before the discrete transition occurs, any delay is enabled (as there is no location invariant on $S0$), so x_1 can take any non negative value. Thus $\{x_1 > 0\} = z_\infty$ is the zone assigned to the initial location. Since it is contained in the existing $\langle S0, z_\infty \rangle$ (the duplicate), $\langle S0, x_1 > 0 \rangle$ (the refined node) can be removed and the analysis of the path continues from the remaining node $\langle S0, z_\infty \rangle$.

The next node to refine is $\langle R0, z_\infty \rangle$. The transition from $\langle S0, z_\infty \rangle$ resets x_1, so its initial value in location $R0$ is 0. The invariant of the location limits the maximum value of x_1, hence the maximum value of a time transition at location $R0$ is 2. Thus the reachable zone in $R0$ satisfies $x_1 \leq 2$. The refinement of the trace continues, and $C1$ turns out to be reachable. The refined zone graph is depicted on Fig **??**.

3.3 Realization

TODO: Megvalósítós részek ide + pszeudokód

3.4 Evaluation

TODO: Terminálódás, komplexitás, stb.

Chapter 4

Implementation

4.1 Environment

TODO: TTMC bemutatása

4.2 Designed modules

TODO: Elkészített modulok, helyük, és használatuk bemutatása

4.3 Measurements

TODO: Mérések készítése, bemutatása, célok ismertetése stb. Ha esetleg a továbbfejlesztések implementálására is marad idő, akkor ez az egész fejezet mehet hátrébb.

Chapter 5

Improvements

5.1 Background

Szerintem ennek itt külön kell lennie, mivel a nagy background részben furán venné ki magát.

5.1.1 Online pathfinding

5.1.2 Activity, Equality

5.2 Improvement

TODO: ötletek felsorolása

5.3 Implementation details

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

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: 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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.