Towards Efficient CEGAR-Based Reachability Analysis of Timed Automata

Rebeka Farkas, András Vörös
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Budapest, Hungary
Email: rebeka.farkas@inf.mit.bme.hu, vori@mit.bme.hu

***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.**

I. 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. Our 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 [3] the abstraction is applied on the locations of the automaton. In [4] the abstraction of a timed automaton is an untimed automaton. In [5], [6], and [7] abstraction is applied on the variables of the automaton.

Our 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: our 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. Section II provides some definitions and basic knowledge about timed automata and CEGAR. In Section III our approach is explained, and a simple implementation is described. Section IV gives some final remarks.

II. Background

In this section we define the important aspects of timed automata and briefly explain the relevant parts of the reachability algorithm presented in [1]. We present the verification of Fischer's protocol as an example. CEGAR is also explained at a high level.

A. 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.

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 $v(c)$ 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 $I(l)$.

B. 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 $\langle l, z'\rangle$. Unfortunately the described graph can possibly be infinite.

A concept called *normalization* is introduced in [1]. Let $k(c)$ denote the greatest value to which clock c is compared. 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 applied on $z^{\uparrow}$ before inclusion is checked. Using normalization the zone graph is finite, but unreachable states may appear.

The operation *split* [1] is introduced to eliminate such states. 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 zone. If the result is a set of zones, then multiple new nodes have to be introduced to the zone graph. Applying split results in a zone graph, that is a correct and finite representation of the state space.

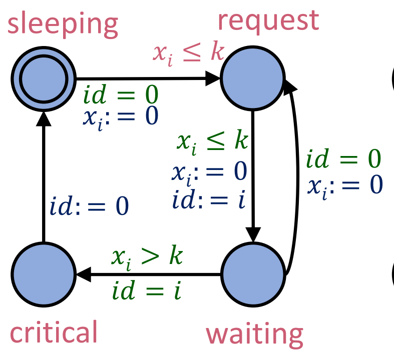

Fig. 1. Fischer's protocol

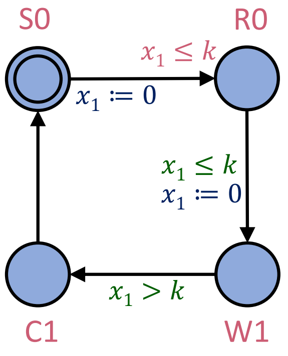

Fig. 2. Timed automaton

Example: Fischer's protocol assures mutual exclusion by bounding the execution times of the instructions. It can be applied to a number of processes accessing a shared variable. Fig. 1 shows the operation of a process. The location *critical* indicates that the process is in the critical section. The value of the shared variable id ranges between 0 and n, where n denotes the number of processes. The model also contains a clock variable x_i for each process where $i \in \{1 \ldots n\}$ denotes the identifier of the process. The constant k is a parameter of the automaton.

The mutual exclusion property would suggest that at any given time at most one of the processes is in the *critical* location. In order to check the given property we must construct a timed automaton that models the operation of a given number of processes.

As our definition of timed automaton only allows clock variables in the system, everything else must be encoded in the location. To demonstrate, Fig. 2 shows the reachable locations of the product automaton of Fischer's protocol where $n = 1$. The names of the locations refer to the original locations of the process, the number denotes the value of the variable id.

C. CEGAR

The CEGAR approach introduced in [2] makes abstraction refinement a key part of model checking. The idea is illustrated on Fig. 3.

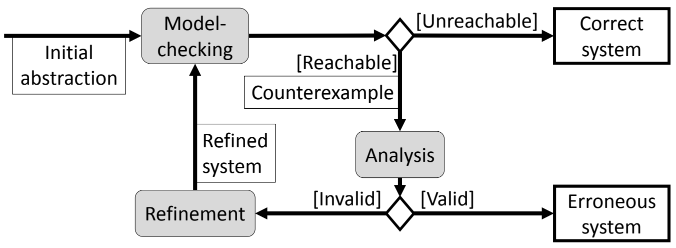

Fig. 3. Counterexample guided abstraction refinement

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.

III. 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.

A. Phases of CEGAR

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

1) Initial Abstraction: The first step of the CEGAR-approach is to construct an initial abstraction, which is an over-approximation 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

$$\begin{tikzcd} \ll S0, z_\infty \gg \arrow[r] & < R0, z_\infty > \arrow[d] \\ < C1, z_\infty > \arrow[u] & < W1, z_\infty > \arrow[l] \end{tikzcd}$$

Fig. 4. The abstract zone graph of the automaton

$$\begin{tikzcd} \ll S0, z_\infty \gg \arrow[r] & < R0, x_1 \leq 2 > \arrow[d] \\ < C1, x_1 > 2 > \arrow[u] & < W1, z_\infty > \arrow[l] \end{tikzcd}$$

Fig. 5. The refined graph

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. 4.

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

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. 4 is denoted by bold arrows.

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 II-B 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. 4 is performed as follows. Refinements starts with the initial node $\langle S0, z_\infty \rangle$. First, we must consider the edge $\langle C1, z_\infty \rangle \rightarrow \langle S0, z_\infty \rangle$. 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 $\langle C1, z_\infty \rangle \rightarrow \langle S0, z_\infty \rangle$ 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 5.

B. Implementation

A simple implementation of the method is explained in this section. Please note that this is still an on-going research, and our current proof-of-concept implementation is far from optimal.

1) Data Structure: The zone graph is represented by an auxiliary graph that can be formally defined as a tuple $\langle N_A, N_R, E^\uparrow, E^\downarrow \rangle$ where $N_A \subseteq L \times \{z_\infty\}$ is the set of abstract nodes, $N_R \subseteq L \times \mathcal{B}(\mathcal{C})$ is the set of refined nodes, $E^\uparrow \subseteq (N_A \times N_A) \cup (N_R \times N_R)$ is the set of upward edges, and $E^\downarrow \subseteq N \times N$ where N denotes $N_A \cup N_R$ is the set of downward edges. The sets $E^\uparrow$ and $E^\downarrow$ are disjoint. $T^\downarrow = (N, E^\downarrow)$ is a tree. The depth of a node n in T is denoted by $d(n)$.

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. In this case $d(n') = d(n) + 1$.

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 node n' where $d(n') < d(n)$ 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, a shorter 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.

Refined nodes appear in the refinement phase. Upward edges can point from abstract to abstract, or from refined to refined node.

2) Applying our Algorithm to the Graph Structure: In our implementation model checking is performed by breadth-first search (BFS). The graph is built until we reach a node containing the target location at some depth (or the location turns out to be unreachable). The trace can be computed by stepping upward on the downward egdes to the root of T.

Before refining a node n, we take care of the upward edges pointing to it. Since the node is about to be refined no upward edge from an abstract zone should point to it. Thus, if there is an edge $n' \rightarrow n \in E^\uparrow$, we remove it, and instead continue BFS from n', until the current depth of T is reached. It can be proven that this will not introduce new counterexamples.

Refinement is performed by replacing abstract nodes with refined ones. Refined zones are calculated as described above. If at some point a node is replicated (because of the split operation), then the subtree should be copied as well. New paths are introduced in the new subtrees, that have to be analyzed later.

If a zone z is a subzone of z' where $d(\langle l, z \rangle) > d(\langle l, z' \rangle)$ we introduce an upward edge $\langle l, z \rangle \rightarrow \langle l, z' \rangle$, and terminate the analysis of the current trace, as it can be proven that it will turn out to be invalid.

IV. Conclusion

This paper introduced a new CEGAR-based reachability analysis of timed automata. Unlike the similar existing approaches, where the refinement phase focuses on the complete automaton, the key idea of our approach is to refine the zone graph incrementally. A prototype implementation is also introduced.

References

[1] J. Bengtsson and W. Yi, “Timed automata: Semantics, algorithms and tools,” in *Lectures on Concurrency and Petri Nets*, ser. LNCS. Springer Berlin Heidelberg, 2004, vol. 3098, pp. 87–124.

[2] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 752–794, 2003.

[3] S. Kemper and A. Platzer, “SAT-based abstraction refinement for real-time systems,” *Electronic Notes in Theoretical Computer Science*, vol. 182, pp. 107–122, 2007.

[4] T. Nagaoka, K. Okano, and S. Kusumoto, “An abstraction refinement technique for timed automata based on counterexample-guided abstraction refinement loop,” *IEICE Transactions*, vol. 93-D, no. 5, pp. 994–1005, 2010.

[5] H. Dierks, S. Kupferschmid, and K. G. Larsen, “Automatic abstraction refinement for timed automata,” in *Formal Modeling and Analysis of Timed Systems, FORMATS’07*, ser. LNCS. Springer, 2007, vol. 4763, pp. 114–129.

[6] F. He, H. Zhu, W. N. N. Hung, X. Song, and M. Gu, “Compositional abstraction refinement for timed systems,” in *Theoretical Aspects of Software Engineering*. IEEE Computer Society, 2010, pp. 168–176.

[7] K. Okano, B. Bordbar, and T. 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.