Activity-Based Abstraction Refinement for Timed Systems

Rebeka Farkas[1], Ákos Hajdu[1,2]
[1]Budapest University of Technology and Economics, Department of Measurement and Information Systems
Email: rebeka.farkas@inf.mit.bme.hu, hajdua@mit.bme.hu
[2]MTA-BME Lendület Cyber-Physical Systems Research Group

***Abstract*—Formal analysis of real time systems is important as they are widely used in safety critical domains. Such systems combine discrete behaviours represented by control states and timed behaviours represented by clock variables. The counterexample-guided abstraction refinement (CEGAR) algorithm utilizes the fundamental technique of abstraction to system verification. We propose a CEGAR-based algorithm for reachability analysis of timed systems. The algorithm is specialized to handle the time related behaviours efficiently by introducing a refinement technique tailored specially to clock variables. The performance of the presented algorithm is demonstrated by runtime measurements on models commonly used for benchmarking such algorithms.**

I. Introduction

Safety critical systems, where failures can result in serious damage, are becoming more and more ubiquitous. Consequently, the importance of using mathematically precise verification techniques during their development is increasing.

Formal verification techniques are able to find design problems from early phases of the development, however, the complexity of safety-critical systems often prevents their successful application. The behaviour of a system is described by the set of states that are reachable during execution (the state space) and formal verification techniques like model checking examine correctness by exploring it explicitly or implicitly. However, the state space can be large or infinite, even for small instances. Thus, selecting appropriate modeling formalisms and efficient verification algorithms is very important. One of the most common formalisms for describing timed systems is the formalism of timed automata that extends finite automata with clock variables to represent the elapse of time.

When applying formal verification, reachability becomes an important aspect – that is, examining whether a given erroneous state is reachable from an initial state. The complexity of the problem is exponential, thus it can rarely be solved for large models. A possible solution to overcome this issue is to use abstraction, which simplifies the problem to be solved by focusing on the relevant information. However, the main difficulty when applying abstraction-based techniques is finding the appropriate precision: if an abstraction is too coarse it may not provide enough information to decide reachability, whereas if it is too fine it may cause complexity problems.

There are several existing approaches in the literature for CEGAR-based verification of timed automata, including [1] where the abstraction is applied on the locations of the automaton, [2] where the abstraction of a timed automaton is an untimed automaton and [3]–[5] where abstraction is applied on the clock 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 to gain a finer state space: our algorithm combines existing approaches with new techniques to create a refinement strategy that increases efficiency by refining the state space directly.

II. Background

A. Timed Automata

Clock variables (*clocks*, for short) are a special type of variables, whose value is constantly and steadily increasing. Naturally, their values can be modified, but the only allowed operation on clock variables is to *reset* them – i.e., to set their value to 0. It's an instantaneous operation, after which the value of the clock will continue to increase.

A *valuation* $v : \mathcal{C} \to \mathbb{R}$ assigns a non-negative real value to each clock variable $c \in \mathcal{C}$, where $\mathcal{C}$ denotes the set of clock variables. In other words a valuation defines the values of the clocks at a given moment of time.

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 $n \in \mathbb{N}$. In other words a clock constraint defines upper and lower bounds on the values of clocks and the differences of clocks. Note, that bounds are always integer numbers. The set of clock constraints are denoted by $\mathcal{B}(\mathcal{C})$.

A *timed automaton* extends a finite automaton with clock variables. It can be defined as a tuple $\mathcal{A} = \langle L, l_0, E, I \rangle$ where

- L is the set of locations (i.e. control states),
- $l_0 \in L$ 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 [6].

The automaton's edges are defined by the source location, the guard (represented by a clock constraint), the set of clocks to reset, and the target location.

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 that modify the state of the automaton. The state $\langle l, v \rangle$ has a *discrete transition* to $\langle l', v' \rangle$ if there is an edge $e(l, g, r, l') \in E$ in the automaton such that v satisfies g, v' assigns 0 to any $c \in r$ and assigns $v(c)$ to any $c \notin r$, and v' satisfies $I(l')$.

The state $\langle l, v \rangle$ has a *time transition* (or *delay*, for short) to $\langle l, v' \rangle$ if v' assigns $v(c) + d$ for some non-negative d to each $c \in \mathcal{C}$ and v' satisfies $I(l)$.

B. Reachability Analysis

In case of timed automata the reachability problem can be defined as follows.

Input: An automaton $\langle L, l_0, E, I \rangle$, and a location $l_{err} \in L$.

Output: An execution trace $\sigma = l_0 \xrightarrow{t_0} l_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} l_{err}$ from l_0 to l_{err} or *No*, if l_{err} is unreachable.

One of the most efficient algorithms for deciding reachability is the one used by *Uppaal*[1], a model checker for timed automata. The core of the algorithm is published in [6]. Before presenting the approach, some basic definitions are provided.

A *zone* z is a set of non-negative clock valuations satisfying a clock constraint. A *zone graph* is a finite graph consisting of $\langle l, z \rangle$ pairs as nodes, where $l \in L$ refers to some location of a timed automaton and z is a zone. Edges represent transitions.

A node $\langle l, z \rangle$ of a zone graph represents all states $\langle l, v \rangle$ where $v \in z$. Since edges of the zone graph denote transitions, a zone graph can be considered as an (exact) abstraction of the state space. The main idea of the algorithm is to explore the zone graph of the automaton, and if a node $\langle l_{err}, z \rangle$ exists in the graph for some $z \neq \emptyset$, l_{err} is reachable, and the execution trace can be provided by some pathfinding algorithm.

The construction of the graph starts with the initial node $\langle l_0, z_0 \rangle$, where l_0 is the initial location and z_0 contains the valuations reachable in the initial location by time transitions. Next, for each outgoing edge e of the initial location (in the automaton) a new node $\langle l, z \rangle$ is created (in the zone graph) with an edge $\langle l_0, z_0 \rangle \to \langle l, z \rangle$, where $\langle l, z \rangle$ contains the states to which the states in $\langle l_0, z_0 \rangle$ have a discrete transition through e. Afterwards z is replaced by $z^{\uparrow}$ where $\langle l, z^{\uparrow} \rangle$ represents the set of all states reachable from a zone $\langle l, z \rangle$ by time transitions. The procedure is repeated on every node of the zone graph. If the states defined by a new node $\langle l, z \rangle$ are all contained in an already existing node $\langle l, z' \rangle$ ($z \subseteq z'$), $\langle l, z \rangle$ can be removed, and the incoming edge can be redirected to $\langle l, z' \rangle$.

Unfortunately, it is possible that the described graph becomes infinite. In order to prevent this, [6] introduces an operation called *normalization* to apply on $z^{\uparrow}$ before inclusion is checked. Let $k(c)$ denote the greatest value to which clock c is compared in the automaton. This operation overapproximates the zone treating the interval $(k(c), \infty)$ as one, abstract value for each $c \in \mathcal{C}$, since for any valuation v such that $v(c) > k(c)$ constraints of the form $c > n$ are satisfied, and constraints of the form $c = n$ or $c < n$ are unsatisfied.

Using normalization the zone graph is finite, and if there are no difference constraints in the automaton, reachability will be decided correctly, however, in case of difference constraints the algorithm may terminate with a false positive result.

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

The operation *split* [6] 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 created in the zone graph (with edges from the original node). Applying split results in a zone graph, that is a correct and finite representation of the state space [6].

Implementation is also provided in [6]. The zones are stored in an $n \times n$ matrix form (the so-called *Difference Bound Matrix*, DBM), where $n = |\mathcal{C}| + 1$, and each row and column represents a clock, except for the first ones that represent the constant 0. An entry $D[i, j] = (m, \prec)$, where $m \in \mathbb{Z} \cup \{\infty\}, \prec \in \{<, \leq\}$ of the DBM D represents the constraint $c_i - c_j \prec m$, where $c_0 = 0$. (It is proven, that all atomic clock constraints can be transformed to this form.) Each entry of a DBM represents the strongest bound that can be derived from the constraints defining the zone.

Pseudocodes are also provided for operations, such as *add()* (adds an atomic constraint to the zone), *reset()* (resets the given clock), *up()* (calculates $z^{\uparrow}$), *norm()* and *split()* to calculate successor states automatically, as well as some additional operations, such as *free()* (removes all constraints on a clock).

C. Activity

The (exact) *activity* abstraction is proposed in [7] to reduce the number of clock variables without affecting the state space. A clock c is considered *active* at some location l (denoted by $c \in Act(l)$) if its value at l may influence the future operation of the system. It might be because c appears in the $I(l)$, or in the guard g of some outgoing edge (l, g, r, l'), or because $c \in Act(l')$ for some l' reachable from l without resetting c.

If $Act(l) < |\mathcal{C}|$ holds for each $l \in L$, the number of clock variables can be reduced by reconstructing the automaton, by removing all $c \notin Act(l)$ and *renaming* $c \in Act(l)$ for each $l \in L$ such that after renaming less clocks remain. This is possible, even if all $c \in \mathcal{C}$ is active in at least one location, since clocks can be renamed differently in distinct locations.

Before presenting how activity is calculated some new notations are introduced. Let $clk : \mathcal{B}(\mathcal{C}) \to 2^{\mathcal{C}}$ assign to each clock constraint the set of clocks appearing in it. Define $clk : L \to 2^{\mathcal{C}}$ such that $c \in clk(l)$ iff $c \in clk(I(l))$ or there exist an edge (l, g, r, l') such that $c \in clk(g)$.

Activity is calculated by an iterative algorithm starting from $Act_0(l) = clk(l)$ for each $l \in L$. In the i^{th} iteration $Act_i(l)$ is derived by extending $Act_{i-1}(l)$ by $Act_{i-1}(l') \setminus r$ for each edge (l, g, r, l'). The algorithm terminates when it reaches a fix point, i.e. when $Act_i(l) = Act_{i-1}(l)$ for each $l \in L$.

D. CEGAR

In order to increase the efficiency of model checking, (approximate) abstraction can be used [8]: a less detailed system model is constructed with a state space overapproximating that of the original one, model checking is applied to this simple model, and if the erroneous state is unreachable in the abstract

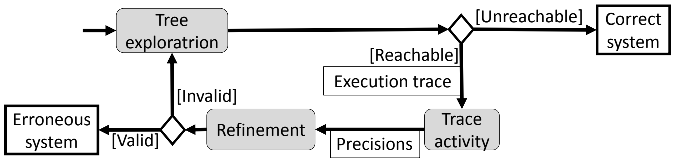

Fig. 1. Algorithm overview

model, the system is correct. Otherwise the model checker produces an abstract counterexample that is examined on the original system, and if it is feasible, the system is incorrect. If it is invalid, the abstraction is too coarse to decide reachability.

Counterexample-guided abstraction refinement (CEGAR) [9] extends this approach into an iterative algorithm, by refining the abstract state space in order to eliminate the invalid counterexample. Model checking is applied on the refined state space (that is still an abstraction of the original one) and the so-called CEGAR-loop starts over.

III. Activity-Based Abstraction Algorithm

The main idea of our new algorithm is to explore the state space without considering clock variables and refining it (calculating zones) trace by trace, based on the discovered counterexamples. Figure 1 depicts the basic operation of the algorithm. Note, that the phases of this algorithm correspond to the phases of CEGAR.

To increase efficiency not all clock variables are included – the relevant clocks for each node in the path (the *precision*) are chosen by an algorithm we have developed based on the one described in Section II-C. To avoid confusion, zones will appear with their precision denoted, e.g. z_C denotes a zone z of precision $C \subseteq \mathcal{C}$.

A. Data structure

In our algorithm the formalism that represents the abstract state space can be defined as a tuple $\langle N_e, N_u, E^{\uparrow}, E^{\downarrow} \rangle$ where

- $N_e \subseteq L \times \mathcal{B}(\mathcal{C})$ is the set of explored nodes,
- $N_u \subseteq L \times \mathcal{B}(\mathcal{C})$ is the set of unexplored nodes,
- $E^{\uparrow} \subseteq (N_e \times N)$, where $N = N_e \cup N_u$ is the set of upward edges and
- $E^{\downarrow} \subseteq (N_e \times N)$ is the set of downward edges.

The sets N_e and N_u as well as the sets $E^{\uparrow}$ and $E^{\downarrow}$ are disjoint. $T^{\downarrow} = (N, E^{\downarrow})$ is a tree.

Nodes are built from a location and a zone and (downward) edges represent transitions like in the zone graph but in this case nodes are distinguished by the trace through which they are reachable. 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. Downward edges have similar roles to edges of the zone graph, while upward edges are used to avoid exploring the same states multiple times. An upward edge from a node n to a previously explored node n' means that the states represented by n are a subset of the states represented by n', thus it is unnecessary to keep searching for a counterexample from n, because if there exists one, another one will exist from n'. Searching for new traces is only continued on nodes without an outgoing upward edge. This way, the graph can be kept finite.

Initially, the graph contains only one, unexplored node $n_0 = \langle l_0, z_{\emptyset} \rangle$, and as the state space is explored, unexplored nodes become explored nodes, new unexplored nodes and edges appear, until a counterexample is found, or there are no remaining unexplored nodes. During the refinement phase zones are calculated, new nodes and edges appear and complete subtrees disappear. State space exploration will then be continued from the unexplored nodes, and so on.

B. State space exploration

State space exploration is performed in the following way. In each iteration a node $n = \langle l, z_C \rangle \in N_u$ is chosen. First, it is checked if the states n represents are included in some other node $n' = \langle l, z'_{C'} \rangle$ where $C = C'$. In this case an upward edge $n \to n'$ is introduced and n becomes explored. Otherwise, n has yet to be explored. For each outgoing edge $e(l, g, r, l')$ of l in the automaton a new node $\langle l', z_{\emptyset} \rangle \in N_u$ is introduced with an edge pointing to it from n, which becomes explored. If any of the new nodes contains l_{err}, the state space exploration phase terminates and the proposed counterexample $\sigma = n_0 \xrightarrow{t_0} n_1 \xrightarrow{t_1} \cdots \xrightarrow{t_n} n_{err} = \langle l_{err}, z_{\emptyset} \rangle$ is the trace reaching n_{err} in $T^{\downarrow}$. Otherwise, another $n \in N_u$ is chosen, and so on.

If the state space is explored and l_{err} does not appear in it, the erroneous states are unreachable, and the system is correct.

C. Trace activity

After finding a possible counterexample the next task is to calculate the necessary precisions. The presented algorithm is a modified version of the one described in Section II-C.

Based on *activity* we introduce a new abstraction $Act_{\sigma}(n)$, called *trace activity* which assigns precisions to nodes on the trace (instead of locations of the automaton), that only include the clocks whose value affects the reachable states of the trace.

Trace activity is calculated iterating backwards on the trace. In the final node n_{err} the valuations are not relevant, as the only question is whether it is reachable – $Act_{\sigma}(n_{err}) = \emptyset$. For $n_i \neq n_{err}$, $Act_{\sigma}(n_i)$ can be calculated from $Act_{\sigma}(n_{i+1})$ and the edge $e_i(l_i, g_i, r_i, l_{i+1})$ used by transition t_i. Since all $c \in r_i$ are reset, their previous values will have no effect on the system's future behaviour – they can be excluded. It is necessary to know if t_i is enabled, so $clk(g_i)$ must be active, as well as $clk(I(l_i))$ since $I(l_i)$ have to be satisfied. This gives us the formula $Act_{\sigma}(n_i) = (Act_{\sigma}(n_{i+1}) \setminus r_i) \cup clk(g_i) \cup clk(I(l_i))$.

D. Refinement

The task of the refinement phase is to assign correct zones of the given precision for each node in the trace and to decide if the counterexample is feasible. It is important to mention that the zones on the trace may already be refined to some precision C' that is independent from the new precision C. In this case the zone has to be refined to the precision $C \cup C'$.

Refinement starts from the initial zone z_0 that can be refined to $z_{C_0} = \bigwedge_{c_i, c_j \in C_0} c_i = c_j$, where C_0 is the required precision. After that z_{C_i} of node n_i on the trace can be calculated from $z_{C_{i-1}}$ of node n_{i-1} with the operations mentioned in Section II-B, with some modifications to handle the precision change.

First, the guard has to be checked. If there are no states in $z_{C_{i-1}}$ that satisfy g_{i-1}, the counterexample is invalid, and the abstract state space has to be refined: since t_{i-1} is not enabled, the corresponding edge, and the belonging subtree has to be removed from the graph, and the algorithm can continue by searching for another counterexample.

Next, the clocks in r_{i-1} are reset. This can be performed using operation *reset()*. Change of precision is also applied at this point. Assume that the precision of the source zone is C_i and the target zone has to be refined to precision C_{i+1}.

Variables $C_{old} = C_i \setminus C_{i+1}$ have to be excluded before executing the transition. Consider the DBM implementation of zones. Excluding the unnecessary clocks from the zone can be performed by *free(c)* for each $c \in C_{old}$, but according to the pseudocode in [6] the operation only affects the row and the column belonging to c. Thus, for space saving purposes, the row and column of c can simply be deleted from the DBM.

Variables $C_{new} = C_{i+1} \setminus C_i$ have to be introduced. *Trace activity* guarantees that clocks are only introduced when they are reset, thus, it can be performed by adding a new row and column to the DBM, that belong to c and calling *reset(c)*.

The next step is to apply the invariant. If this results in an empty zone, the transition is not enabled – the subtree has to be deleted, and the algorithm continues by searching for another counterexample. Otherwise, *up(), split()*, and *norm()* has to be applied to calculate the precise zone (or zones).

The node n_i can be refined by replacing the current zone with the calculated one, however, the incoming upward edges have to be considered first. An edge $n \to n_i \in E^{\uparrow}$ means n_i represents all states that n represents – this may not be true after the refinement. Thus, the upcoming edges are removed and their sources are marked *unexplored*.

It is important to consider that sometimes the *split()* operation results in more than one zones. Similarly to the case of the zone graph, this can be handled by replicating the node. Refinement has to be continued from each new node, thus the refinement of a trace may introduce new subtrees. The tree structure allows this, however, it is important to mark the new nodes *unexplored*, since only the outgoing edges representing the transition on the trace are created, the other possible outgoing edges have yet to be explored.

IV. Evaluation

We evaluated the performance of the presented algorithm with measurements. The inputs are scalable automata chosen from Uppaal's benchmark data[2] that is widely used for comparing the efficiency of such algorithms. Network automata with discrete variables were unfolded to timed automata before the measurements. The results are depicted in Table I.

[2]https://www.it.uu.se/research/group/darts/uppaal/benchmarks/

TABLE I
Measurement results (ms)

CSMA2	CSMA3	CSMA4	Fisch2	Fisch3
264	1 113.5	9 808	292	5 650
Token8	Token32	Token128	Token512	Token2048
838	2 173	4 966	12 580	100 892

The models are denoted by CSMAn, Fisch$n$, and Token$n$ for the CSMA/CD, Fischer and Token ring/FDDI protocols of n participants, respectively. The Token ring protocol is a special input, since the examined safety property can be proven solely based on the structure of the automaton, thus the analysis of the initial abstraction is able to prove the property. This proves how useful abstraction is, but the measurements on this automaton can only demonstrate the efficiency of the pathfinding algorithm, which turned out to be $\mathcal{O}(n^2)$. Memory problems occurred at the Fischer protocol of four processes and the CSMA/CD protocol of five stations. For smaller instances the algorithm always terminated with the expected result.

V. Conclusions

This paper provided a CEGAR-based algorithm for reachability analysis of timed automata, that applies abstraction on the zone graph, and calculates the required precision for the refinement using *trace activity*. The efficiency of the algorithm was demonstrated by measurements. Results suggest that the pathfinding algorithm is efficient, but the memory usage has yet to improve.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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*. Washington - Brussels - Tokyo: IEEE, Dec. 1996, pp. 73–81.

[8] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, Sep. 1994.

[9] 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.