Automatic Abstraction Refinement for Timed Automata*

Henning Dierks[1], Sebastian Kupferschmid[2], and Kim G. Larsen[3]

[1] OFFIS, Oldenburg, Germany
dierks@offis.de
[2] University of Freiburg, Germany
kupfersc@informatik.uni-freiburg.de
[3] Aalborg University, Denmark
kgl@cs.auc.dk

Abstract. We present a fully automatic approach for counterexample guided abstraction refinement of real-time systems modelled in a subset of timed automata. Our approach is implemented in the MOBY/RT tool environment, which is a CASE tool for embedded system specifications. Verification in MOBY/RT is done by constructing abstractions of the semantics in terms of timed automata which are fed into the model checker UPPAAL. Since the abstractions are over-approximations, absence of abstract counterexamples implies a valid result for the full model. Our new approach deals with the situation in which an abstract counterexample is found by UPPAAL. The generated abstract counterexample is used to construct either a concrete counterexample for the full model or to identify a slightly refined abstraction in which the found spurious counterexample cannot occur anymore. Hence, the approach allows for a fully automatic abstraction refinement loop starting from the coarsest abstraction towards an abstraction for which a valid verification result is found. Nontrivial case studies demonstrate that this approach computes small abstractions fast without any user interaction.

1 Introduction

Embedded systems often control safety critical systems. Hence, formal methods are mandatory to establish correctness results for such systems. Since model checking is a technique that does the analysis without any user interaction it is widely examined in the literature and many tools are available. However, model checking suffers from the so called state space explosion problem, i. e., the complexity of the verification procedure is exponential in the size of the system. Therefore, several techniques have been invented to tackle this problem like symbolic representation and abstractions. In most cases the requirements of embedded systems refer to the timing. Verification of real-time systems by model checking is even more difficult because time adds another source of complexity. As a consequence, checking a model that represents a nontrivial real-time system always requires to find an appropriate abstraction that can be checked

* This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See http://www.avacs.org/ for more information.

J.-F. Raskin and P.S. Thiagarajan (Eds.): FORMATS 2007, LNCS 4763, pp. 114–129, 2007.

within the given resources of memory and time. If the used abstraction is an over-approximation, then the absence of an abstract counterexample implies the absence of concrete counterexamples. Such an abstraction is called a *safe* abstraction. However, if a model checker analyses a safe abstraction and discovers a counterexample, the question arises whether the counterexample is spurious or not, i. e., whether there is a concrete counterpart in the full model.

Without automation, model checking by abstractions consists of several, error-prone and time-consuming tasks. First, the user has to select a proper initial abstraction which usually requires a deep knowledge of the system under consideration. If the model checker finds a counter example in this abstraction, then the user has to analyse whether it is spurious or not. If the counterexample is real, then the user is done, otherwise the selected abstraction was too coarse. In this case the initial abstraction must be refined so that the spurious counterexample is eliminated. These steps are repeated until there is no abstract counterexample, or the abstract counterexample is also a counterexample for the full system.

The approach presented in this paper automates *all* steps of the abstraction refinement loop in the setting of real-time specifications given as PLC automata [1]. This leads to a fully automated abstraction refinement loop as depicted in Fig. 1. This loop starts with the coarsest possible abstraction and iterates as long as spurious counterexamples are found. If the model checker finds an abstract counterexample, then a counterexample analyser is used to check whether it is spurious. Our analyser first constructs a test automaton from the abstract counterexample. The composition of the full model with the newly generated automaton is then fed into the model checker. We extended the model checker in such a way that if the counterexample is spurious, the model checker also reports hints why it is spurious. These hints are then used in the refinement step in order to eliminate the abstract counterexample. The termination of this abstraction refinement loop is guaranteed by the fact that each iteration removes at least one of finitely many abstracted variables.

To demonstrate the potential of our approach we carried out several nontrivial case studies, i. e., the respective full models cannot be handled within the given memory resources. Our approach is able to handle them with only a fraction of the resources, starting from the initial abstraction towards a refined abstraction for which a definite answer could be found.

The remainder of the paper is structured as follows: the next section briefly introduces the formalisms we work with, Section 3 shows how we check for spuriousness

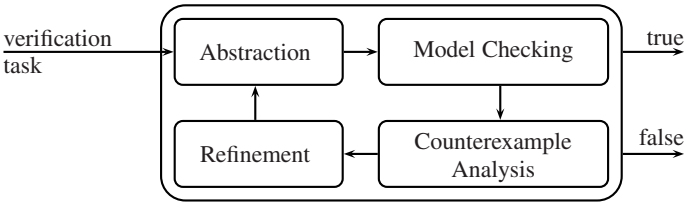

Fig. 1. Counterexample guided abstraction refinement loop

of counterexamples. Section 4 describes our used abstraction refinement cycle and the algorithms to provide the necessary information. Section 5 provides experimental evaluation of the implementation of the method within UPPAAL. In Section 6 we discuss related work and Section 7 concludes.

2 Preliminaries

In our setting, a verification task consists of a real-time system, which is given in terms of PLC automata, together with a temporal property to verify. To solve such a verification task with our framework, we use MOBY/RT to compute an abstraction of the given PLC automata. MOBY/RT is a tool for the development and analysis of PLC automata [2]. From the verification perspective the most important feature of MOBY/RT is the generation of safe abstractions of an arbitrary set of PLC automata together with a temporal property into the input syntax of UPPAAL 3.4. These abstractions are generated according to the entities (e. g., variables and delays) of the PLC automata the user has chosen for abstraction.

In the following we will briefly describe PLC automata, timed automata and the relation between these two formalisms.

2.1 PLC Automata

The formal specification language called PLC automata has been developed to enable formal verification of real-time properties of PLC programs. A Programmable Logic Controller (PLC) is a standardised hardware platform which is especially equipped to simplify the design of real-time controllers in practice. It can be seen as a simple computer with a special real-time operating system. A PLC communicates with the environment via unbuffered asynchronous input and output channels. The environment may change the values of the inputs arbitrarily whereas the outputs are controlled by the PLC. PLCs behave in a cyclic manner where every cycle consists of the following three phases: first the inputs are polled, then the new output values are computed and finally the outputs are updated. The repeated execution of this cycle is managed by the operating system. The only part the programmer has to adapt is the computing phase. Depending on the program and on the number of inputs and outputs there is an upper time bound for such a cycle.

In the definition of PLC automata we consider the upper time bound for a complete cycle and the possibility to delay the system's reactions depending on state and input. Figure 2 gives an example of a PLC automaton. The automaton has three locations q_0, q_1, q_2 and an output variable `output` that ranges over `ok`, `test` and `alarm`. It reacts to a Boolean input variable `signal`. Every state has two labels shown below its name in the picture. They define a delay time d and a constraint S on the input. The value of d defines the *minimal* amount of time that the system should stay in the corresponding state provided that, meanwhile, only input values satisfying S are polled.

A PLC automaton describes the behaviour of the system in the computation phase. The operational behaviour is similar to a finite state machine, i. e., depending on the polled input value the system changes both its state and its output. The behaviour is

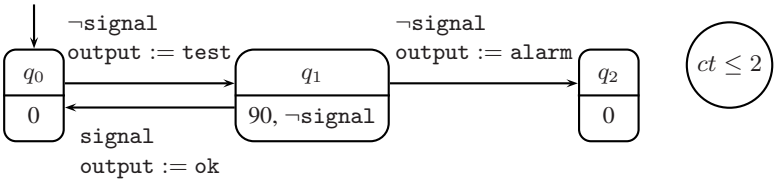

Fig. 2. An example of a PLC automaton

modified in only one case: if the annotations of the current state are d and S, then a transition is only executed when the polled input does not satisfy S *or* the current state holds longer than d time units. That means a transition is disabled if the polled input satisfies S and the current state has not exceeded the delay time d.

Thus, the PLC automaton in Fig. 2 behaves as follows: it starts in state q_0 and remains there as long as it reads only the input `signal`. The first time it reads `¬signal` it changes to state q_1. In q_1 the automaton reacts to the input value `signal` by changing the state back to q_0 *independently* of the time it stayed in state q_1. If it reads `¬signal` in q_1 the behaviour of the system depends on the duration it already stayed in q_1. If 90 time units have elapsed it can take the transition to q_2. Otherwise, no transition is fired. In q_2 the automaton remains forever. Hence, we know that the automaton changes its output to `alarm` when `¬signal` holds a little bit longer than 90 time units because the cycle time (2 time units) has to be considered.

Note that PLC automata are implementable. In [3] a translation of PLC automata into source code for PLCs is given. In [2] also a translation into C++ code (tailored to Lego Mindstorms) has been developed. The intended logical relationship between the execution of the code by the real-world hardware and the semantics given in the rest of this paper is *refinement*. In other words: the real-world implementation cannot show a behaviour that is not covered by the formal semantics.

2.2 UPPAAL and Timed Automata

UPPAAL[1] is a modelling, simulation, and verification tool for real-time systems modelled as networks of extended timed automata [4, 5]. We assume the reader is roughly familiar with timed automata and their commonly used extensions. Therefore, we only provide a short description of timed automata and the extensions which we use for the construction of test automata. For more details about this tool the reader is referred to the UPPAAL tutorial [6].

Figure 3 shows a network of three parallel timed automata P, Q and R as it is used in UPPAAL. The system has three clocks x, y and z, three integer variables k, m and n, binary synchronisation labels a and b and a committed location q_2 (indicated by the "c:"). The initial state of the overall system is given by the three initial locations p_1, q_1 and r_1 together with the initial values of the integer variables and the initial values for the clocks. All clocks and integer values are 0 in this state. The bounded integer variables are part of each system state, they can occur in transition guards and can be changed with the assignment of a transition. Two edges of different automata can fire

[1] See `http://www.uppaal.com/`

a synchronised transition of the system if they are labelled with complementary synchronisation labels (represented by "!" and "?" respectively). Suppose the system is in a state where it is in p_2 and q_1 and the guards of the outgoing edges of these two locations are enabled, then the system can take a synchronised transition, which leads to a state where P is in p_3 and Q is in q_2. For an edge annotated with a synchronisation label it is not possible to fire a transition without a synchronisation partner. Edges without synchronisation label are called τ-transitions. A system state is a committed state if at least one of the current automata locations is committed. Committed locations are a mechanism to restrict the behaviour of the overall system. A committed state cannot delay and the next transition applied to this state must involve an outgoing edge of a committed location. For example, if the system is in a state where P is at p_1, Q is at the committed location q_2 and R is at r_1, then this system state is committed. The system then has to take the edge from q_2 to q_3 which requires Q to synchronise with R. The location p_1 is labelled with a location invariant. This restricts the duration the system can idle in the current state. Suppose the system is in a state where P is at p_1 and the value of the clock x is 0, then the system can remain at most 11 time units before leaving this state.

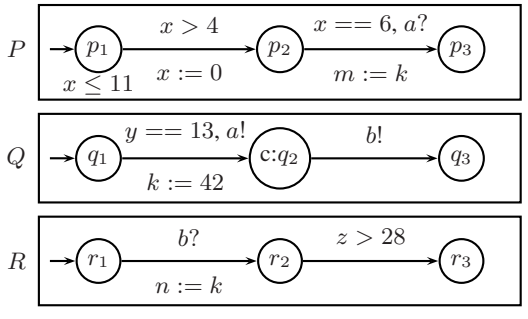

Fig. 3. A simple UPPAAL model

2.3 Timed Automata Semantics of PLC Automata

The semantics of PLC automata is defined in terms of timed automata [3, 1]. Due to space limitations we can just present a sketch using the example of Fig. 2. The semantics of this automaton, depicted in Fig. 4, consists of two timed automata and the following (global) variables and clocks: z is a clock that represents the duration of the PLC cycle, y is a clock that measures how long the system stays in q_1, `Out` and `sig` represent the variables `Output` resp. `signal` used in the PLC automaton and `Psig` is a variable that represents the *polled* value of the input variable `signal`. The states of the PLC automaton appear as a set of locations in the timed automaton. For example, the state q_0 has two representatives in the timed automaton in order to represent the internal state of the PLC within the cycle (q_0/p stands for polling, q_0/cu stands for computing and updating). The transitions between q_0/p and q_0/cu implement the polling behaviour of the PLC automaton in state q_0. The polling step copies the current value of `sig` to the variable `Psig` which is used for the subsequent computations. The outgoing transitions from q_0/cu represent the reactions of the PLC automaton in state q_0.

Depending on the polled value of `sig` the system switches to q_1 or remains in q_0. Moreover, these transitions reset the z clock because the cycle has been finished. If the automaton switches to q_1 the output variable is changed appropriately and the clock y is reset to be able to check whether 90 time units have elapsed while staying in state q_1. Since q_1 is equipped with a delay the semantics needs more locations to represent the behaviour. After polling (transition from q_1/p to q_1/c) the timed automaton checks whether the delay time has passed or the delay condition is not satisfied. If this is the case, then a transition to q_1/u is enabled. Otherwise the system can switch to q_1/d ("delayed") where the cycle is finished. Note that the semantics is nondeterministic with respect to the timing in order to model the physical reality. To ensure progress each state has the invariant $z \leq 2$. Therefore, the timed automaton has to execute a cycle within the upper bound because only transitions to q_i/p reset the z clock. To model the environment that can change the input variable `sig` arbitrarily a driver automaton is added that can always toggle the input.

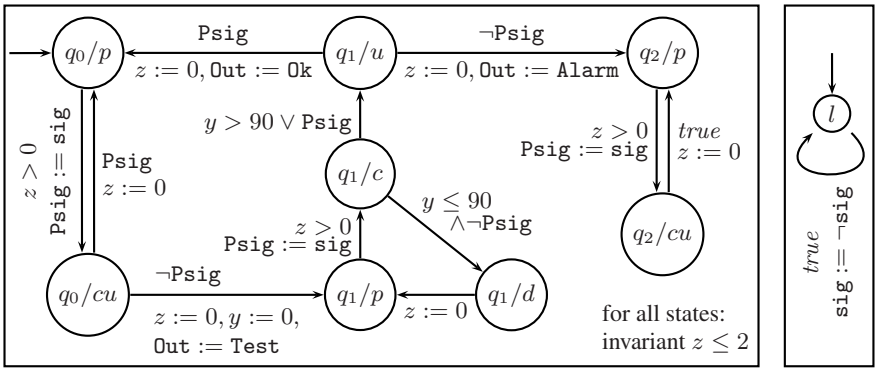

Fig. 4. Semantics of the PLC automaton from Fig. 2 in terms of timed automata

Abstractions of this semantics are generated by selecting a set of variables and clocks. Then the abstract semantics is derived from the full semantics by removing all assignments to the abstracted variables and clocks and replacing all constraints by the strongest constraint that is weaker than the original and that does not contain abstracted entities. For example, the guard $y \leq 90 \wedge$ `Psig` is replaced by `Psig` if the clock y is abstracted and `sig` is not.

3 Counterexample Analysis

In the abstraction refinement loop, when model checking a safety property of an abstract system returns a counterexample, one has to investigate whether it is spurious or not. One possible way of doing this is to build a *linear* test automaton T, in which every location has at most one outgoing edge, from the abstract counterexample. This test automaton is then composed with the full system F to be verified. If the last location of the test automaton is reachable in the composed system, then we know that the abstract counterexample is also a counterexample for the full system.

It is desirable that the test automaton T is able to find a concretisation if there is any and, moreover, it is mandatory that T restricts the behaviour of the full system as much as possible. If this is not the case, the test automaton is useless because it would make no difference to model check the full system only. We therefore construct the test automaton so that it synchronises with the full model as often as possible. By doing this, only a small fraction of the state space of the full model is explored. This makes perfect sense, because we are only interested in a concretisation of the abstract trace.

A UPPAAL counterexample is a finite sequence of states that are either connected via transitions or via delays with a certain duration. For example, for the timed automata system given in Fig. 3 and the temporal formula $\mathrm{E}\Diamond(p_3 \wedge q_3 \wedge r_3)$, UPPAAL reports the error trace given in Fig. 5. The trace starts with the initial state of the system and ends with a state satisfying the given property. Each state of the trace assigns each automaton of the system a location, each variable a value of the variable's domain and each clock a rational value. A test automaton built from such a trace proceeds if the full model of the system executes a transition that enables the guard of the next transition of the test automaton. The problem is to decide whether a step in the full model matches. Checking the values of the variables is straight-forward. To check the clock values is a bit tricky because a trace may also have clock valuations with rational numbers. Below we show how this problem can be solved. Another problem is to match the locations. The locations are not subject to abstractions. Hence, they appear in both the full and the abstract model. However, UPPAAL does not provide any syntactic means to refer to locations in guards. Therefore, we introduce an auxiliary integer variable for each automaton of the system. The current value of this variable identifies uniquely the current location of the automaton. Thus, the test automaton is able to match locations by checking the corresponding auxiliary variable. In order to preserve the same behaviour with respect to global time we add a clock *time* to the test automaton. This clock is never reset and therefore it represents the current duration of the trace at any time. The next three paragraphs give a detailed description on how to build a test automaton.

The first element of an abstract trace is the initial state of the abstract system. Although we do not expect any differences with the full model here, we add a corresponding check for reasons of completeness. We add a transition leading from the initial location t_0 of the test automaton to a new location t_1. The guard of this transition checks if the system's variables have the right values at time point $time = 0$. To restrict the full model's behaviour t_0 is a committed location.

A delay transition with duration $d \in \mathbb{Q}$ of the abstract counterexample is translated as follows: suppose that, after the transition is made, the system is in a state which is described by the valuation val and that $T \in \mathbb{Q}$ is the sum of all durations that occur before the transition. Let the most recently added location of the test automaton be t_n. In this case we add two locations t_{n+1} and t_{n+2} to the test automaton and the two transitions $t_n \xrightarrow{await(T+d)} t_{n+1}$ and $t_{n+1} \xrightarrow{await(T+d) \wedge check(val)} t_{n+2}$. Here $await(q)$ for $q \in \mathbb{N}$ is $time = q$ and for $q \in \mathbb{Q} \setminus \mathbb{N}$ is $\lfloor q \rfloor > time \wedge time < \lceil q \rceil$. Note that by the definition of $await(q)$ the test automaton searches for an over-approximation of the given abstract trace. The expression $check(val)$ is the conjunction of tests whether all discrete variables of val equal the valuation of the state. Note that this approach is correct because if there is no concretisation found using this over-approximation then there is no concretisation at

```
( P.p1 Q.q1 R.r1 ) x=0 y=0 z=0 k=0 m=0 n=0
   Delay: 7
( P.p1 Q.q1 R.r1 ) x=7 y=7 z=7 k=0 m=0 n=0
   P.p1->P.p2  x > 4, tau, x := 0
( P.p2 Q.q1 R.r1 ) x=0 y=7 z=7 k=0 m=0 n=0
   Delay: 6
( P.p2 Q.q1 R.r1 ) x=6 y=13 z=13 k=0 m=0 n=0
   Q.q1->Q.q2  y == 13, a!, k := 42
   P.p2->P.p3  x == 6, a?, m := k
( P.p3 Q.q2 R.r1 ) x=6 y=13 z=13 k=42 m=42 n=0
   Q.q2->Q.q3  1, b!, 1
   R.r1->R.r2  1, b?, n := k
( P.p3 Q.q3 R.r2 ) x=6 y=13 z=13 k=42 m=42 n=42
   Delay: 15.5
( P.p3 Q.q3 R.r2 ) x=21.5 y=28.5 z=28.5 k=42 m=42 n=42
   R.r2->R.r3  z > 28, tau, 1
( P.p3 Q.q3 R.r3 ) x=21.5 y=28.5 z=28.5 k=42 m=42 n=42
   Delay: 0.5
( P.p3 Q.q3 R.r3 ) x=22 y=29 z=29 k=42 m=42 n=42
```

Fig. 5. Trace reported by UPPAAL for the query $\mathrm{E}\Diamond(p_3 \wedge q_3 \wedge r_3)$ of the system from Fig. 3

all. If a concretisation is found then the concrete trace may differ from the abstract trace with respect to the non-integer delays but it is a trace of the full model that satisfies the reachability property. An alternative approach would be to multiply all timing constants with the common denominator and check for equality only.

A τ-transition in the counterexample also introduces two locations in the test automaton. Suppose the last added location of the test automaton is t_n and after the τ-transition the system is in a state described by the valuation val. Let T and t_n be defined as above. Here, we add the locations t_{n+1} and t_{n+2} to the test automaton and the two transitions $t_n \xrightarrow{await(T) \wedge C?} t_{n+1}$ and $t_{n+1} \xrightarrow{await(T) \wedge check(val)} t_{n+2}$. Note that we exploit the fact that the abstract model made a τ-transition to introduce a synchronisation of the test automaton with the full model via the channel C. By this, the test automaton is notified as soon as the full model changed the values for the variables and the search space is reduced. Since time is supposed to pass, the location t_n cannot be committed. However, to restrict the behaviour of the system t_{n+1} can be committed.

A synchronised transition of the counterexample is translated as follows: suppose again, that the valuation val, T and t_n are defined as above. In contrast to how we handle a τ-transition, we cannot force the test automaton to participate. However, we know that whenever MOBY/RT generates a synchronised transition this only happens directly after a τ-transition and that all synchronised transitions are leaving a committed location. Therefore we add the location t_{n+1} and the transition $t_n \xrightarrow{check(val) \wedge await(T)} t_{n+1}$. As synchronised transitions can happen sequentially when more than one automaton is in the network and the transition's origin is a committed location we have to make location t_n committed, too.

Consider the trace generated by UPPAAL in Fig. 5 again. If we apply the above construction rules we get the test automaton described in Fig. 6.

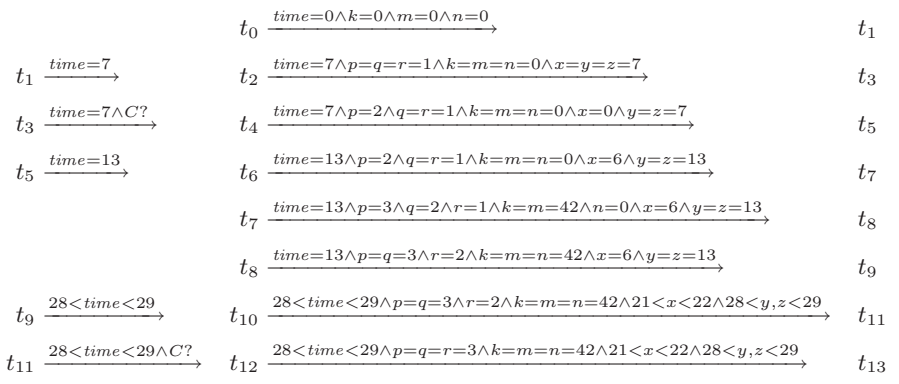

Fig. 6. Test automaton for the trace in Fig. 5

4 Abstraction Refinement

When an abstract counterexample reveals to be spurious, a model checker normally does not give any hint why this is the case. Usually, the next steps one has to do in order to get a refined version of the current abstraction is to analyse the counterexample. Here, one has to identify variables or clocks respectively that hinder the progress of the test automaton. Normally, this has to be done *manually* and, on the one hand, is a tedious and time consuming procedure and on the other hand requires a deep understanding of the model to verify. We will henceforth refer to integer variables and clocks as variables respectively.

Our approach automates the analysis of the counterexample. We did this by extending UPPAAL so that if an abstract counterexample is spurious, UPPAAL reports this *and* at the same time provides a set of variables that should not be abstracted in the next iteration of the abstraction refinement loop.

To determine a refined abstraction, we exploit the fact that our test automata are linear. If the abstract counterexample turns out to be spurious, then there is a unique transition in the test automaton whose starting location is reached, but not its target location. We call the starting location of this transition the *dead end location* and the transition itself we call the *dead end transition*. The dead end transition can either be blocked because there is no enabled transition in the full system that can synchronise with the test automaton, or when there is no reachable state with a valuation of the variables that satisfies the guard of the dead end transition.

Our approach determines a minimal set of variables u_{hint}, so that if these variables had different values, the test automaton could take at least the dead end transition. This is done on the fly, while UPPAAL checks if the error trace is spurious. Figure 7 sketches UPPAAL's verification algorithm for safety properties. The arguments of the *verify* function are the initial state of the system s_0 and the property ϕ to verify. We extended this algorithm by including the lines 13–15.

During the analysis, UPPAAL checks each outgoing transition from a location in the current state s if it is enabled. If this is the case, the successor state s' is computed

```
1 function verify(s_0, φ):
2   open = {s_0}, closed = ∅
3   while open ≠ ∅:
4     s = open.pop()
5     if s ⊨ φ:
6       return True
7     if s ∉ closed:
8       closed.push(s)
9       for each outgoing transition t of s:
10        if t is enabled:
11          s' = succ(s, t)
12          open.push(s')
13          progress(s')
14        else:
15          analyse(s, t)
16  return False
```

Fig. 7. Reachability analysis

and added to the open list. In addition to the normal verification function, we now call *progress* with the successor state s' (line 13). Pseudo-code for the *progress* function is given in Fig. 8. It determines the reached location of the test automaton which has currently the smallest distance to the test automaton's last location. Remember, if the last location of the test automaton is reachable, then the counterexample is a real counterexample. After the execution of the *verify* function $dead\_end$ is the dead end location.

```
1 function progress(s):
2   if dist(s(test)) < dist(dead_end):
3     dead_end = s(test)
4     u_hint = ∅
```

Fig. 8. On the fly detection of the dead end location

If, during the generation of successor states, a transition t is not enabled it is passed together with its starting state s to the *analyse* function (line 15). Pseudo code for this function is shown in Figure 9. The *analyse* function checks if the test automaton in s is in the current $dead\_end$ location. If this is the case, then it checks if applying t would enable the current dead end transition t_{dead_end}. If this is the case, then *analyse* collects all the variables and clocks respectively that appear in unsatisfied constraints of t's guard. If the set of these variables u is smaller than u_{hint}, then u_{hint} is updated. After the execution of the *verify* function u_{hint} contains variables that hinder the execution of the dead end transition.

After UPPAAL has checked that the counterexample is spurious, all variables that occur in the set of unsatisfied constraints u_{hint} are reported. These variables should not be abstracted in the next iteration, as they hinder the progress of the test automaton. This ensures that the revealed spurious counterexample will not be found in the next iteration. In the following, we explain that the reported variables are likely to be helpful.

```
1  function analyse(s, t):
2    if s(test) ≠ dead_end:
3      return
4    if t_dead_end is synchronised:
5      if t can synchronise with t_dead_end:
6        u = {c ∈ inv(succ(s, t)) | c unsat constraint}∪
7              {c ∈ guard(t) | c unsat constraint}
8        if |u| ≤ |u_hint| ∨ u_hint = ∅:
9          u_hint = u
10   else if assignment of t makes guard(t_dead_end) True:
11     u = {c ∈ inv(succ(s, t)) | c unsat constraint}∪
12           {c ∈ guard(t) | c unsat constraint}
13     if |u| ≤ |u_hint| ∨ u_hint = ∅:
14       u_hint = u
```

Fig. 9. On the fly extraction of least blocking variables. Used expressions: $\mathrm{inv}(s)$: conjunction of s's location invariants, $s(test)$: the location of the test automaton in s, $\mathrm{succ}(s,\, t)$: the successor state of s reached through t, $\mathrm{guard}(t)$: t's guard, $\mathrm{dist}(l)$: distance from a location l of the test automaton to the last location of the test automaton in terms of transitions.

From the construction of the test automaton we know that there are at most two types of transitions in the test automaton: synchronised transitions and τ-transitions. The guard of such a synchronised transition is always satisfiable because it was already satisfied when the starting location of the transition was reached. It only checks that no time elapses since the last transition. Depending on which part the transitions represent from the abstract counterexample, we can distinguish three different cases.

If the dead end transition belongs to one of the transitions introduced for a delay in the abstract counterexample, then the progress of the test automaton is blocked because the full system cannot idle due to an unsatisfied location invariant. This is only possible if this location invariant talks about a clock that was abstracted away because in the abstraction it is possible to take this transition. Therefore this clock should not be abstracted in the next iteration of the abstraction refinement loop.

If the dead end transition belongs to one of the two transitions introduced for a τ-transition in the abstract counterexample, then we know that the progress of the test automaton stops because of a τ-transition in the full system. The reason for this is that the clock guards of the two introduced transitions are satisfied. So the only reason why this may block is that the guard that checks the valuation of the variables is not satisfied. But this is only possible if there is no enabled τ-transition in the full system whose assignments would make the guard true. As there is such a transition in the abstraction, we again know that this must be because of an unsatisfied transition guard. So the variables that occur in the unsatisfied constraints of this guard should not be abstracted in the next iteration.

The last possible reason why the dead end transition is blocked is that there is no enabled transition with an assignment that would make the guard of the dead end transition true. From the construction of the test automaton this can only be a synchronised transition in the full system because a τ-transition in the full system always has to synchronise with the test automaton which is not possible. As we know that there is

a synchronised transition in the abstract system that makes the guard of the dead end transition true, this transition would do it also in the full system. The reason why the progress of the test automaton is stopped is that either the guard of this synchronised transition or a location invariant of the successor state is not satisfied.

5 Experiments

To demonstrate the potential of our approach we chose the "Single-tracked Line Segment" (SLS) case study which stems from the UniForM-project [7]. It is the specification of a control system for a single-tracked line segment for tramways. It is implemented by distributed PLC automata [8]. We took three different models of the SLS case study [8] as examples. As the safety property to verify, we chose the mutual exclusion of drive permissions, i. e., the control system never gives permission to both directions simultaneously.

The first model (M1) we checked is a manipulated system that we obtained by changing a delay time but with the assumption that everything is implemented on only one hardware device. The full UPPAAL model we got from MOBY/RT had 9 processes, 2 clocks and 24 integer variables. Table 1 shows in the first row the resources needed to check the full model of M1 using the standard UPPAAL verification engine. It took 310 seconds to verify that the manipulated delay time does not lead to an error if the system is implemented on one device. In the following rows the steps of the abstraction refinement loop are given. Each step consists of a verification run to find an abstract counterexample (left columns) and the check for spuriousness (right columns). For these runs we use a variant of UPPAAL called UPPAAL/DMC that allows for *directed model checking* [9, 10]. Directed model checking is the application of heuristics to model checking and was pioneered a few years ago by Edelkamp et al. [11, 12], christening this research direction directed model checking. The use of directed model checking makes sense because it can be expected that the current abstraction contains (abstract) counterexamples and directed model checking detects counterexamples faster. Note that whenever a spurious counterexample is found a refined abstraction is derived. This refined abstraction considers more entities (at least one clock or one integer variable more than before).

For M1 it turns out that with our counterexample guided abstraction refinement we can prove correctness of the model using an abstraction with 1 clock, 4 processes and 14 integer variables less than the full model. The required memory is about 5 % and the *summarized* time consumption is approx. 3 % compared to the full model. Note that in abstraction 3, the number of processes has changed. The reason for this is that whenever a variable is added to the next abstraction that is triggered by the environment, then an additional automaton is added to the system that drives this variable. These driver automata are automatically generated by MOBY/RT.

For the next verification problem we removed the assumption about the partitioning of the PLC automata onto hardware devices. The second experiment (M2) represents a distributed system. Now, the manipulated delay time leads to an incorrect system. However, it was not possible to find a counterexample in the full model within the given memory limit of 2 GBs. This time the abstraction refinement loop had to iterate 8 times to generate an abstraction for which a definite answer was found, i. e., a counterexample

Table 1. Abstraction refinement results for the experiments. Abbreviations: #c: number of clocks, # p: number of processes, # v: number of integer variables, time: runtime in seconds, mem: memory peak in MB, trace: length of found error trace, CE: counterexample.

Model	# c	# p	# v	time	mem	trace	# c	# p	# v	time	mem	result
M1: full							2	9	24	310.0	721	verified
Abstr. #1	0	4	3	0.0	8	20	3	10	24	0.0	7	spurious CE
Abstr. #2	1	4	3	0.0	9	22	3	10	24	3.2	36	spurious CE
Abstr. #3	1	5	5	0.0	9	23	3	10	24	0.4	11	spurious CE
Abstr. #4	1	5	6	0.0	9	23	3	10	24	1.3	20	spurious CE
Abstr. #5	1	5	8	0.0	9	34	3	10	24	2.2	27	spurious CE
Abstr. #6	1	5	10	0.8	9	–						verified
M2: full							3	10	25	> 1527.0	> 2048	out of memory
Abstr. #1	0	5	3	0.0	8	27	4	11	25	0.0	8	spurious CE
Abstr. #2	1	5	3	0.0	9	29	4	11	25	0.0	8	spurious CE
Abstr. #3	2	5	3	0.0	9	30	4	11	25	113.6	491	spurious CE
Abstr. #4	2	5	6	0.1	9	88	4	11	25	44.6	247	spurious CE
Abstr. #5	2	6	8	0.4	9	64	4	11	25	7.6	64	spurious CE
Abstr. #6	2	6	9	0.1	9	44	4	11	25	14.3	108	spurious CE
Abstr. #7	2	6	11	0.2	9	62	4	11	25	15.8	97	spurious CE
Abstr. #8	3	6	11	0.2	9	77	4	11	25	0.5	12	disproved
M3: full							3	10	25	> 1242.0	> 2048	out of memory
Abstr. #1	0	5	3	0.0	8	27	4	11	25	0.0	8	spurious CE
Abstr. #2	1	5	3	0.0	9	29	4	11	25	0.0	8	spurious CE
Abstr. #3	2	5	3	0.0	9	30	4	11	25	117.8	753	spurious CE
Abstr. #4	2	5	6	0.1	9	88	4	11	25	45.3	312	spurious CE
Abstr. #5	2	6	8	0.4	9	64	4	11	25	7.7	80	spurious CE
Abstr. #6	2	6	9	0.1	9	44	4	11	25	14.5	141	spurious CE
Abstr. #7	2	6	11	0.2	9	62	4	11	25	15.9	130	spurious CE
Abstr. #8	3	6	11	3.3	13	–						verified

in the full model. The computed abstraction saved 4 processes and 14 integer variables. The memory and time consumption was *at most* 25 % respectively 14 % compared to the full model.

In our final experiment (M3) we reverted to the original delay time. Again, it was not possible to check the full model within the memory limits. The abstraction refinement loop generated the same sequence of refined abstractions and terminated after 8 iterations again. But this time UPPAAL was able to verify that the final abstraction has no counterexample.

All these experiments show that the abstraction refinement presented in this paper is able to generate abstractions effectively for which definite verification results can be found. The main benefits are that there is no need for human interaction at all, an abstraction of the model is computed automatically for which a reliable verification result can be computed and that this approach reduces significantly the resources (time and memory). However, there is no guarantee that this approach computes a *minimal* abstraction but it is obvious that it will terminate since each iteration adds at least one of the finitely many entities of the model.

6 Related Work

Abstraction refinement was pioneered by Clarke et al. [13] in the early 90s. Since then many researchers have automated this process starting with the work of Balarin and Sangiovanni-Vincentelli [14]. The term counterexample guided abstraction refinement was coined by Clarke et al. [15]. This work deals with discrete timed systems and ACTL* formulas. It has been extended to continuous-time models in [16, 17]. The main idea of these approaches is to refine the discrete state space by an appropriate replication of states which have been involved in a spurious counterexample to avoid this trace in the next iteration.

Alur et al. [18] have proposed a predicate abstraction based approach for a counterexample guided abstraction refinement procedure applicable to hybrid systems. The refined abstraction extends the state space by predicates describing additional information on the continuous state space. These additional predicates are constructed by identifying the dead-end state of a spurious trace and an analysis which polyhedron in the continuous state space avoid a reoccurrence of the trace.

A similar approach was proposed by Segelken [19]. Here the analysis of the spurious trace generates an automaton that is put in parallel with the model in the next iteration of the abstraction refinement loop. This automaton represents an infeasible fragment of the previous spurious trace. By the construction of the automaton this infeasible fragment is avoided.

Also in the area of timed automata there are approaches for model checking by iterative refinement of approximations. One of these approaches was implemented in Laroussinie's and Larsen's compositional model checker CMC [20]. This tool starts with a small subset of the automata of the system. It subsequently adds automata to this set and minimises the intermediate result.

Another abstraction refinement approach was developed by Sorea et al. [21, 22] in which predicate abstraction was used on the level of the regions which are defined by predicates over clocks. The approach uses symbolic counterexamples from failed model-checking attempts. Such a symbolic counterexample represents a sequence of sets of states, and can be seen as generalisation of a linear counterexample. To exclude a spurious symbolic counterexamples from further iterations, new abstraction predicates are chosen randomly from a set of predefined predicates. Except for the fact that new abstraction predicates are chosen randomly, this approach is, in some respect, quite similar to what we are proposing here. The main differences are the nature of the counterexamples and that new abstraction variables are selected more carefully. Unfortunately the authors do not give any runtime results.

7 Conclusion and Future Work

We presented an approach for counterexample guided abstraction refinement for a subclass of timed automata. In this paper we defined how to construct test automata that can be used to check whether a full model is able to behave as the abstract trace, i. e., to check whether an abstract trace is spurious or not. Moreover, we extended the model checker UPPAAL in such a way that it executes an analysis of *why* a full model cannot

execute a spurious trace. The result of this analysis is used to refine the given abstraction in a way that the spurious counterexample cannot occur anymore. This approach enabled us to construct a closed abstraction refinement loop in which verification of a system starts with the coarsest abstraction and with each iteration the abstraction is refined until a result is found that holds for the full model, too.

In its current version our approach is able to refine the abstraction by adding variables or clocks. From our point of view, the most promising direction of future work is to extend the approach such that it also refines the set of automata considered in the abstraction. At present the set of PLC automata is fixed. However, if a system consists of many parallel components it makes sense to start with a small subset thereof. Since the semantics of a subset is an over-approximation this fits our approach. Then the abstraction refinement analysis needs to be extended accordingly in a way that it can also identify PLC automata for the abstraction refinement. Having such an extension the abstraction refinement loop would start with the coarsest abstraction of only those automata that manipulate variables appearing in the requirement.

In this paper we have presented and implemented an abstraction refinement methodology for PLC automata. However, the methodology is generally applicable to the full range of timed automata based models expressible within UPPAAL. Here, a particularly challenge will be the generalisation of the automated analysis of counterexamples presented in Section 4 to deal with the rich imperative language (including structured data types, user-defined types as well as user-defined functions and procedures) provided in UPPAAL 4.0. We envisage the need for incorporating the work of Sørensen and Trane on slicing UPPAAL 4.0 models [23].

References

1. Dierks, H.: Time, Abstraction and Heuristics – Automatic Verification and Planning of Timed Systems using Abstraction and Heuristics. Technical report, University of Oldenburg (2006)
2. Olderog, E.R., Dierks, H.: Moby/RT: A Tool for Specification and Verification of Real-Time Systems. J. UCS 9, 88–105 (2003)
3. Dierks, H.: Specification and Verification of Polling Real-Time Systems. PhD thesis, University of Oldenburg (1999)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126, 183–235 (1994)
5. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: Paterson, M.S. (ed.) Automata, Languages and Programming. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
6. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
7. Krieg-Brückner, B., Peleska, J., Olderog, E.R., Baer, A.: The uniform workbench, a universal development environment for formal methods. In: Woodcock, J.C.P., Davies, J., Wing, J.M. (eds.) FM 1999. LNCS, vol. 1709, pp. 1186–1205. Springer, Heidelberg (1999)
8. Dierks, H.: PLC-Automata: A New Class of Implementable Real-Time Automata. Theor. Comput. Sci. 253, 61–93 (2001)
9. Kupferschmid, S., Hoffmann, J., Dierks, H., Behrmann, G.: Adapting an AI planning heuristic for directed model checking. In: Valmari, A. (ed.) Model Checking Software. LNCS, vol. 3925, pp. 35–52. Springer, Heidelberg (2006)

10. Kupferschmid, S., Dräger, K., Hoffmann, J., Finkbeiner, B., Dierks, H., Podelski, A., Behrmann, G.: UPPAAL/DMC – Abstraction-based Heuristics for Directed Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 679–682. Springer, Heidelberg (2007)
11. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit-state model checking in the validation of communication protocols. STTT (2004)
12. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Directed explicit model checking with HSF-SPIN. In: Dwyer, M.B. (ed.) Model Checking Software. LNCS, vol. 2057, pp. 57–79. Springer, Heidelberg (2001)
13. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16, 1512–1542 (1994)
14. Balarin, F., Sangiovanni-Vincentelli, A.L.: An iterative approach to language containment. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 29–40. Springer, Heidelberg (1993)
15. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
16. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and Counterexample-guided Refinement in Model-Checking of Hybrid Systems. Int. J. Found. Comput. Sci. 14, 583–604 (2003)
17. Fehnker, A., Clarke, E., Jha, S., Krogh, B.: Refining Abstractions of Hybrid Systems using Counterexample Fragments. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, Springer, Heidelberg (2005)
18. Alur, R., Dang, T., Ivancic, F.: Predicate Abstraction for Reachability Analysis of Hybrid Systems. Trans. on Embedded Computing Sys. 5, 152–199 (2006)
19. Segelken, M.: Abstraction and Counterexample-guided Construction of Omega-Automata for Model Checking of Step-discrete linear Hybrid Models. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, Springer, Heidelberg (2007)
20. Laroussinie, F., Larsen, K.G.: CMC: A tool for compositional model-checking of real-time systems. In: Proc. FORTE/PSTV, pp. 439–456. Kluwer Academic Publishers, Dordrecht (1998)
21. Möller, M.O., Rueß, H., Sorea, M.: Predicate abstraction for dense real-time system. In: Proc. TPTS, Elsevier, Amsterdam (2002)
22. Sorea, M.: Lazy approximation for dense real-time systems. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 363–378. Springer, Heidelberg (2004)
23. Sørensen, U., Trane, C.: Optimization for the Uppaal verification tool. Technical report, Aalborg University (2007)