Synthia: Verification and Synthesis for Timed Automata*

Hans-Jörg Peter[1], Rüdiger Ehlers[1], and Robert Mattmüller[2]

[1] Reactive Systems Group
Saarland University, Germany
[2] Foundations of Artificial Intelligence Group
Freiburg University, Germany

Abstract. We present Synthia, a new tool for the verification and synthesis of open real-time systems modeled as timed automata. The key novelty of Synthia is the underlying abstraction refinement approach [5] that combines the efficient symbolic treatment of timing information by difference bound matrices (DBMs) with the usage of binary decision diagrams (BDDs) for the discrete parts of the system description. Our experiments show that the analysis of both closed and open systems greatly benefits from identifying large relevant and irrelevant system parts on coarse abstractions early in the solution process. Synthia is licensed under the GNU GPL and available from our website.

1 Introduction

A crucial factor for the acceptance of automatic system analysis techniques is how well they scale when the models become more complex. A powerful concept aiming at increasing the scalability is *automatic abstraction refinement*, where, beginning with a coarse abstraction of the original system, only those parts are incrementally refined that are necessary for proving a certain property.

In this paper, we report on Synthia, a new tool that makes abstraction refinement available for the analysis of open real-time systems modeled in a syntactically enhanced variant of the popular timed automata formalism by Alur and Dill [1]. An open system distinguishes between external and internal nondeterminism, of which one type represents an unpredictable environment and the other type represents a partial implementation. Synthia checks if the system is realizable (i.e., whether there exists a full implementation) such that, independent of the environment, some safety requirements are satisfied. Synthia can also certify the (un)realizability by generating a controller that represents safe (violating) implementations (environments). The verification of closed systems, where the implementation is deterministic and complete, is a special case and can equally well be handled by Synthia.

* This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

2 Underlying Approach

Computational model. We interpret a given open real-time system as a two-player game played on a timed automaton, in which player Adam controls the choices of the partial implementation and player Eve controls the nondeterminism of the unpredictable environment. A specified safety property defines the dual winning conditions for the players: while Eve wins whenever the property is eventually violated, Adam wins when the property is always satisfied.

As fundamental computation model, we consider timed game automata [6], an alternating extension of timed automata [1], where the controllability of each transition is assigned to a particular player. We always assume our timed game automata to be strongly nonzeno. SYNTHIA also supports additional syntactic features such as parallel composition or arithmetic expressions over bounded integer variables. In the following, we use the term location to refer to any pure discrete state information, including integer variable valuations. We consider an asymmetric semantics, where Eve is prioritized in situations in which both players can play an active move, which leads to determinacy of all games.

Synthesizing a safety controller corresponds to computing a winning strategy for Adam. For obtaining such a strategy, we compute the set of states from which the players can enforce their respective winning objectives. For Eve, this is done by taking the so-called *attractor* of the set of bad (i.e., requirement-violating) states. Any strategy that enforces staying in the complement of this set is then winning for Adam. In case of a closed system, there are only Eve moves, in which case game solving naturally boils down to checking reachability.

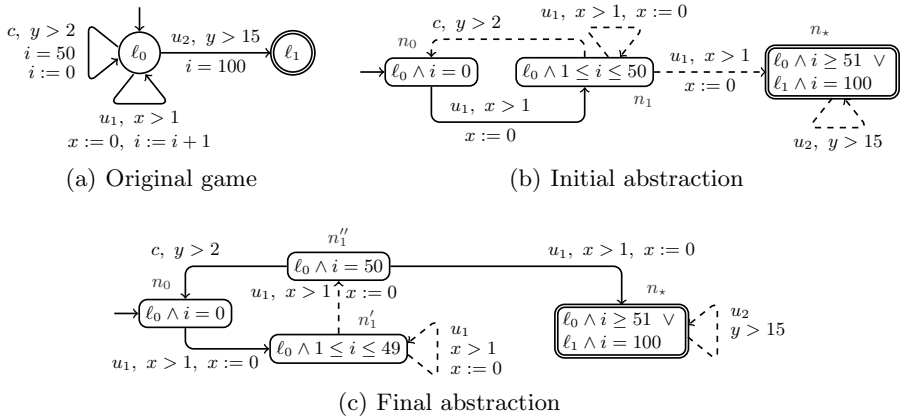
Abstraction refinement. SYNTHIA's main analysis algorithm is an efficient implementation of the approach introduced in [5]. We collapse sets of concrete locations of the original timed game automaton into single abstract locations. In these so-obtained *syntactic* abstractions, we distinguish between *may* and *must* transitions: between two abstract locations n and n', for a player p,

- there is a *may* transition for p, if there is some concrete location subsumed by n having a p-transition to some concrete location subsumed by n';
- there is a *must* transition for p, if all concrete locations subsumed by n have a p-transition to some concrete location subsumed by n'.

We obtain an abstract game by letting Eve play on her must transitions and Adam play on his may transitions. Clearly, in our abstractions Eve is weakened and Adam is strengthened, compared to the original game. Thus, computing the attractor of the bad states in the abstract game yields an under-approximation of the attractor in the original game.

The abstraction refinement procedure begins with the trivial abstraction that comprises (at most) four abstract locations: (1) one subsuming all initial locations; (2) one subsuming all bad locations; (3) one subsuming all safe locations (from which no bad location is reachable); (4) one subsuming all other locations which are not in (1)-(3). Then, in each iteration of the following refinement loop, we compute the attractor of the bad states in the respective current abstraction;[3]

[3] In fact, SYNTHIA incrementally *updates* an attractor under-approximation.



(a) Original game

(b) Initial abstraction

(c) Final abstraction

Fig. 1. Example timed game and its abstractions with transitions surely available (solid lines) and potentially available (dashed lines).

if it contains a concrete initial state, we can stop the iteration as we can surely deduce a concrete winning strategy for Eve. On the other hand, if the abstract game is safe, we refine the abstraction such that the attractor is extended in the refined game. If no such refinement is found, we can deduce that Eve loses and Adam wins.

Optimizations. The symbolic treatment of the locations allows us to apply computationally cheap but effective optimizations based on pure discrete analyses of the control structure of the original game. For instance, in the initial abstraction, Synthia only considers those locations which are both forward reachable from the initial locations and backward reachable from the bad locations. Also, before constructing the initial abstraction, Synthia enlarges the set of bad locations by those locations from which Eve can force the system into a bad location. Furthermore, the selection of possible refinements can be restricted to (an over-approximation of) the forward reachable states.

Example. Consider the timed game depicted in Fig. 1(a) comprising the clocks x and y, as well as the integer variable i ranging from 0 to 100. Adam controls the transition c and Eve controls the transitions u_1 and u_2. Eve wins when ℓ_1 is eventually visited. It is easy to see that Adam has a winning strategy by playing c whenever it is available (i.e., when $i = 50$). The initial abstraction (Fig. 1(b)) is based on all locations that are reachable in a purely discrete manner. Here, we note that the abstract bad location $n_\star$ subsumes the concrete location $\ell_1 \wedge i = 100$ as well as $\ell_0 \wedge i \geq 51$, which is the result of the bad-location enlargement. As this abstract game is safe, the refinement heuristic determines to split n_1 such that u_1 becomes available for Eve, which leads to an enlargement of the attractor. The resulting game (Fig. 1(c)) is still safe as the attractor can only be updated for n_1'' to $y - 1 \leq x \vee 1 < x$. Now, any further refinement would not enlarge the

attractor (e.g., making the u_1-transition between n_1' and n_1'' available to Eve is useless since then, n_1'' is only entered when $x = 0$ and $y > 2$, in which case only c is available but not u_1).

3 The Tool Synthia

Availability and usage. Synthia is licensed under the GNU General Public License and available at

http://react.cs.uni-saarland.de/tools/synthia.

Due to the lack of space, it is impossible to explain all of Synthia's features in this paper. Instead, in this section we present some standard usage scenarios. A detailed description of the command line parameters, the file format, as well as a step-by-step tutorial can be found on the tool's website.

A specification is given in form of an XML file and essentially comprises a plant model with requirements. Assuming that `robot.xml` represents a specification, then the simplest way to execute Synthia is the following:

```
$ synthia  robot.xml
```

This lets Synthia check whether there exists a controller influencing the plant such that regardless of the uncontrollable behavior, the requirements are always satisfied. Specifications can have parameters with default values which can be overridden using the command line argument `-D`:

```
-Dprocesses:2  -Ddelay:23  -Dtimeout:42
```

Requirements are given as conjunctions of assumptions and guarantees. A system does not satisfy its requirements if (1) there is a trace that eventually violates the guarantees, and (2) each prefix of that trace satisfies the assumptions. For example, the following lines of an XML specification file encode a requirement describing a location invariant and a bounded reachability guarantee:

```
<assume>in(loc) imply (x <= {delay})</assume>
<guarantee>(not in(goal)) imply (y <= {timeout})</guarantee>
```

To let Synthia synthesize a controller in addition to checking realizability, the following command line parameters can be used:

```
$ synthia  robot.xml  --synth-cont controller.xml
$ synthia  robot.xml  --synth-cont-plant controlled_plant.xml
```

The former call generates a model (in the Synthia file format) that only comprises the controller, while the latter generates a model where the synthesized controller is embedded into the original plant.

Implementation details. Synthia is written in C++ and uses, besides some standard Boost libraries, the CUDD BDD library [7] for representing transition relations and sets of locations, as well as the Uppaal DBM library [4] for representing and manipulating clock zones.

After parsing the specification, as explained in [5], SYNTHIA constructs a BDD-based representation of the control structure and sets up the initial abstraction. As an extension to [5], SYNTHIA also considers abstractions of the guards of abstract transitions. Hence, a refinement either consists of splitting an abstract location or of making a guard of an abstract transition precise.

The actual game solving procedure that runs on the abstract games and updates the attractor under-approximation is implemented as a pure backward solving algorithm. The selection of refinements is carried out in form of a forward zone-based reachability analysis: only those abstract locations and transitions are considered which appear in this analysis.

As a further optimization, additionally to under-approximating the attractor of the requirement-violating states, SYNTHIA also computes an under-approximation of the attractor of the safe states. This is done by a concurrent game solving procedure, in which Adam is weakened and Eve is strengthened.

4 Experimental Results

Table 1 shows a comparison of SYNTHIA 1.2.0 with the game solver UPPAAL-TIGA 0.16 [2]. We note that the latter subsumes the basic model checking engine of UPPAAL 4.1.4 [3], which is automatically applied in case of closed-system properties.

From left to right, the columns show the benchmark instance, whether it is a safe/realizable instance, the number of refinement steps, the number of abstract locations in the final abstraction, SYNTHIA's running time and memory consumption, the parameters for which UPPAAL-TIGA showed the best results, UPPAAL-TIGA's number of explored states, running time and memory consumption. Running times are given in seconds, memory consumption in MB, the time limit was set to 4 hours, and the memory limit was set to 4 GB. All experiments were conducted on a 2.6 GHz AMD Opteron computer running Ubuntu 10.04. The model files used for the benchmarks can be downloaded along with the tool.

Fischer and *CSMA/CD* are standard benchmarks from the closed-system verification domain. The instances are parametrized in the number of components. The benchmark *Robot* is to decide whether a robot has a strategy to quickly traverse a square-shaped grid with a wall in the middle that has two gates through which the robot can pass. Up to a certain amount of time, nondeterministically, one of the gates can close upon which the robot has to react. The instances are parametrized in the edge length of the grid. The benchmark *Tank* asks for the existence of a controller that controls the inflow to a water tank such that a desired fill level is reached within a given amount of time. It is parametrized by the precision in which the continuous flow is digitized.

Except for unsafe Fischer, where a depth-first-search on the precise system quickly detects the error, SYNTHIA's abstraction refinement approach always clearly outperforms UPPAAL-TIGA. Interestingly, while UPPAAL-TIGA suffers from an exponential blow-up for increasing instance sizes, the final abstractions found by SYNTHIA are several orders of magnitude smaller than the original

Table 1. Performance comparison of SYNTHIA with UPPAAL-TIGA.

	Safe /	SYNTHIA				UPPAAL-TIGA			
Benchmark	Realizable	Steps	Abs	Time	Mem	Params	States	Time	Mem
Fischer 60	No	116	119	2957	1321	-o1	520	5	39
Fischer 65	No	126	129	2247	935	-o1	6	1	26
Fischer 70	No	TIMEOUT				-o1	256	4	41
Fischer 13	Yes	169	172	4	86	-C -S2	29122758	1324	1127
Fischer 14	Yes	196	199	5	88	-C -S2	93835680	4661	3501
Fischer 15	Yes	225	228	6	90	MEMOUT			
Fischer 30	Yes	900	903	355	228	MEMOUT			
Fischer 40	Yes	1600	1603	2628	453	MEMOUT			
Fischer 51	Yes	2601	2604	14262	1405	MEMOUT			
Fischer 52	Yes	TIMEOUT				MEMOUT			
CSMA/CD 15	Yes	4	5	6	118	-C	11681796	442	2639
CSMA/CD 16	Yes	4	5	8	158	-S2	27901956	1302	3072
CSMA/CD 17	Yes	4	5	16	252	MEMOUT			
CSMA/CD 21	Yes	4	5	1474	3804	MEMOUT			
CSMA/CD 22	Yes	MEMOUT				MEMOUT			
Robot 300	No	356	360	10	87		43147361	486	120
Robot 500	No	596	600	58	98		199601281	2322	233
Robot 1000	No	1196	1200	182	145	TIMEOUT			
Robot 2000	No	2396	2400	1153	365	TIMEOUT			
Robot 3000	No	TIMEOUT				MEMOUT			
Robot 300	Yes	376	380	18	97		92655832	1067	165
Robot 500	Yes	626	630	132	129		429037638	4965	356
Robot 1000	Yes	1251	1255	401	256	TIMEOUT			
Robot 2000	Yes	2501	2505	8011	1061	MEMOUT			
Robot 3000	Yes	TIMEOUT				MEMOUT			
Tank 100	No	28	19	3	91		85852	183	416
Tank 300	No	28	19	14	125	-F1	512034	1532	1309
Tank 500	No	28	19	32	176	-F1	1993710	13991	3882
Tank 1000	No	28	19	50	280	MEMOUT			
Tank 5000	No	28	19	1724	996	MEMOUT			
Tank 10000	No	28	19	8077	1923	MEMOUT			
Tank 10	Yes	54	35	1	80		482227	55	280
Tank 20	Yes	42	29	2	82		1978965	449	1668
Tank 30	Yes	45	31	2	84	MEMOUT			
Tank 100	Yes	55	36	8	113	TIMEOUT			
Tank 500	Yes	44	31	51	205	MEMOUT			
Tank 1000	Yes	53	35	107	359	MEMOUT			
Tank 5000	Yes	45	32	1865	1354	MEMOUT			
Tank 10000	Yes	53	35	8349	3808	MEMOUT			

systems: quadratic in the number of components for Fischer, linear in the edge length of the grid for Robot, or even of constant size for CSMA/CD and Tank.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theo. Comp. Sci. **126**(2) (1994)
2. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for playing games! In: CAV. (2007)
3. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Developing uppaal over 15 years. Softw., Pract. Exper. **41**(2) (2011) 133–142
4. David, A.: UPPAAL DBM Library release 2.0.8 (2011)
5. Ehlers, R., Mattmüller, R., Peter, H.J.: Combining symbolic representations for solving timed games. In: FORMATS. (2010)
6. Maler, O., Pnueli, A., Sifakis, J.: On the synthesis of discrete controllers for timed systems (an extended abstract). In: STACS. (1995)
7. Somenzi, F.: CUDD: CU Decision Diagram package release 2.4.2 (2009)