# Towards Generation of Adaptive Test Cases from Partial Models of Determinized Timed Automata

(short paper)

Bernhard K. Aichernig
Florian Lorber
Institute for Software Technology, Graz University of Technology, Austria
{aichernig,florber}@ist.tugraz.at

***Abstract*—The well-defined timed semantics of timed automata as specification models provide huge advantages for the verification and validation of real-time systems. Thus, timed automata have already been applied in many different areas, including model-based testing. Unfortunately, there is one drawback in using timed automata for test-case generation: if they contain non-determinism or silent transitions, the problem of language inclusion between timed automata becomes undecidable. In recent work, we developed and implemented a technique to determinize timed automata up to a certain depth $k$. The resulting timed automata are unfolded to directed acyclic graphs (*DAGs*) up to depth $k$. It was hardly surprising, that the unfolding caused an exponential state-space explosion. Consequently, the language inclusion algorithm we had developed for test-case generation from deterministic timed automata did not scale anymore.**

**Within this paper we investigate how to split the determinized DAGs into partial models, to overcome the problems caused by the increased state space and find effective ways to use the deterministic DAGs for model-based test case generation.**

## I. Introduction

Real-time requirements cause severe challenges for the verification and validation of time-dependent systems. One of the most common formalisms for specifying timed systems are timed automata. They were introduced by Alur et al. in 1994 [3] and in the years since there has been ongoing work in both the theoretical and the practical aspects of timed automata, including the topic of model-based test case generation [8].

Model-based mutation testing is a specific type of model-based testing, in which faults are deliberately injected into the specification model. The aim of mutation-based testing techniques is to generate test cases that can detect the injected errors, if they violate the conformance relation. This means that a generated test case shall fail if it is executed on a system-under-test that implements the faulty model. The power of this testing approach is that it can guarantee the absence of certain specific faults. In practice, it will be combined with standard techniques, e.g. with random test-case generation.

We already used timed automata for model-based mutation testing, using $tioco$ as conformance relation between the specification and the mutants and detecting conformance violations via language inclusion [2]. The approach was limited to deterministic timed automata without silent transitions, as non-determinism might lead our language inclusion to spurious counter examples. This also limited the approach to using one monolithic timed automaton, as the communication of networks of timed automata introduces silent transitions.

In recent work, we found a way to remove silent transitons and determinize timed automata, by unfolding the automata and bounding the length of the traces we investigate. We recently provided a technical report [7] with more information about this determinization approach. The downside to this technique is an exponential state-space explosion caused by the unfolding. Consider the example presented in Figure 1. It is the timed automata specification of a Car Alarm System that consists of four communicating timed automata. The first two automata handle the locks and the doors. If an input is triggered, they pass on an internal signal to the third automaton, that monitors the locks and doors, to arm the system, if the doors are closed and locked for twenty seconds. The last automaton handles the activation and manual or time-triggered deactivation of the alarms, that are triggered if the doors are violently opened while the system is armed. The network contains four inputs (*lock, unlock, close, open*), five internal signals (*locked, unlocked, closed, opened*) that become hidden after building the product, and six output signals (*soundOn, soundOff, flashOn, flashOff, armedOn, armedOff*). Altogether, the example contains only 23 locations. Yet, unfolding its product to the observable depth two already creates 11 locations, without taking into account the locations that can only be reached by traces ending with internal transitions. Unfolding it one step further creates a total of 50 locations. The number of locations grows exponentially, e.g. on depth 12 it is already higher then three million locations.

At this point, applying our previous language inclusion is not feasible anymore and can not be used for the language inclusion check. In this paper, we propose two methods that avoid the generation of the whole DAG and uses meaningful sub-DAGs instead. The first method applies the mutation to the unfolded DAG, instead of applying it to the original specification, as done in our initial process. Consequently, the position of the mutation in the DAG is known, and the check whether the mutation violates the $tioco$ conformance has to be applied only to the sub-DAG beneath.

The second method is based on heuristic based pruning of the DAG. Due to the fact that we are using $tioco$, which supports partial models, pruning away transitions with input labels leads to legal partial models. The test cases that can

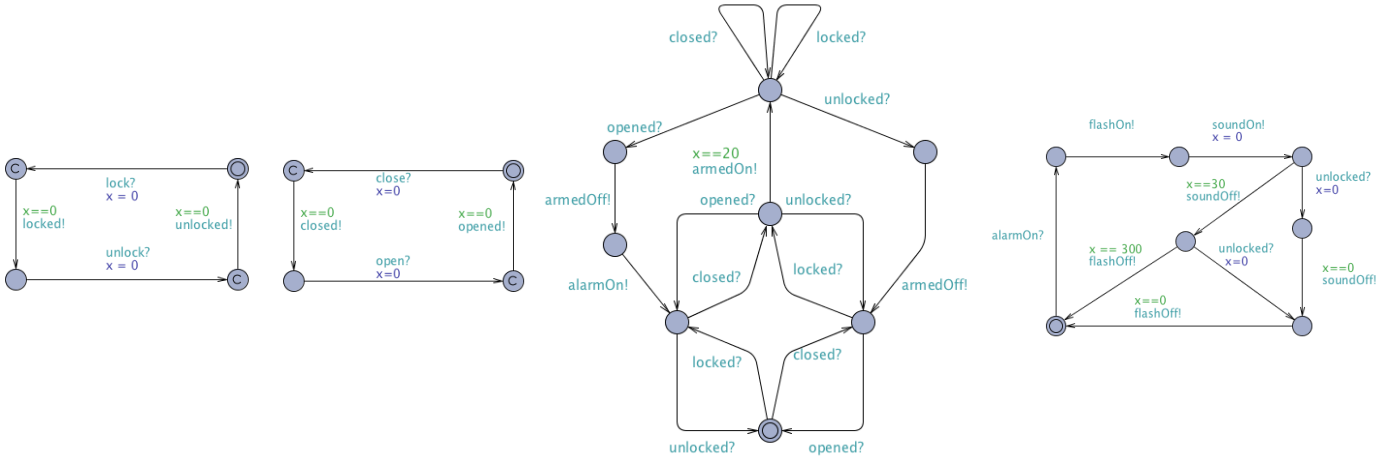


Fig. 1. A network of four automata, specifying a car alarm system: The first two automata handle the locks and doors, the third triggers the arming of the system and the fourth handles the alarms.

be received via pruning the system are far more adaptive than the tests we generated in our previous test case generation approach, as they still contain every trace that can be invoked by the remaining inputs and do not lead to *inconclusive* verdicts.

The rest of the paper is structured as follows: In the next section we will give a very brief introduction to timed automata, stating the properties we assume for the automata we investigate. Then in Section III we explain model-based mutation testing, again stating the problems caused by an increased state space. In Section IV we illustrate our first approach for selecting partial models, where we only select the DAG affected by the mutation. Then in Section V we define the second approach, that works via underspecification of inputs. In Section VI we position ourselves in the context of related work and finally, in Section VII we conclude our paper and give an overview of future steps.

## II. Preliminaries

In this paper we consider timed automata with inputs and outputs (*TAIOs*). Timed automata are finite automata extended by real-valued clocks to measure the time that passes in the locations. Each transition in a timed automata may consist of a label, a set of clocks that are reset if the transition is traversed and a guard over the clocks, specifying the time span in which the transition is enabled. In the case of TAIOS, the labels are divided into inputs, denoted by question marks, and outputs, denoted by exclamation marks. The automata we consider in the following sections are the result of our determinization approach [7] and satisfy the following properties:

They do not contain silent transitons and are deterministic, so each run through the automata invokes the same trace.

They are in the form of directed acyclic graphs, thus each trace starts at the same initial state, and leads to a leaf after at most $k$ steps.

They are single, monolithic timed automata. Even if the original specification was a network of timed automata, a product of all its automata built and unfolded during the determinization approach.

## III. Model-based Mutation Testing

In MBT, the system-under-test is seen as black box and a specification model formalizing the requirements is used in order to decide, wether the system conforms to the specification. The specification model is explored in order to generate test cases, to satisfy a predefined coverage criterion.

In contrast to classic approaches, which use coverage criteria like transition coverage, model-based mutation testing is a particular instance of fault-oriented testing where the test cases are generated in a way that attempts to steer the system-under-test towards intentionally introduced faults. Hence, the idea behind this approach is to generate a set of mutated models, that reflects common modeling errors. If a mutated model does not conform to the original specification, the mutation introduces enabled traces which were not in the original model and these traces serve as the basis to generate a test case. In case that the mutated model conforms to its original version, the mutation does not introduce new behavior with respect to the original specification, hence no useful test case is generated. It follows that test cases are generated only if the mutated model does not conform to its original version.

In previous work [2] we already developed a methodology for model-based mutation testing from timed automata. We developed a *tioco*-conformance check between the specification model and the mutated model via language inclusion. While the mutation was allowed to introduce non-determinism, the original specification was strictly required to be deterministic. This was a severe drawback for applying the approach to real industrial case studies, which was supposed to be revoked after our determinization process.

Our determinization process needs to alter guards of transitions that follow after a silent transition or a non-determinism. Thus it can not be applied to general timed automata, but only to such with bounded traces. Therefore, specification models that are not bounded need to be unfolded before determinization. Due to the state space explosion during the unfolding, the existing test case generator is not able to process our examples within a reasonable time span.

We found some easy adjustments in our algorithms, to

handle the DAGs in a more efficient way. The most valuable adjustment was building a separate step relation formula for every depth of the DAG, taking only those transitions into account, that were on the same depth. Yet, the effect of these adjustments was marginal, compared to the exponential increase in runtime.

## IV. Mutation on the DAG

Originally, it was planned to mutate the original specification before unrolling, thus avoiding the state-space explosion in the mutants. The whole test-case generation process is illustrated in Figure 2 (a).

However, the unfolded structure can also be utilized to improve the mutation and language inclusion process: The unfolding already enumerates the whole state space that is needed for the test-case generation. By mutating the unfolded specification, this information can be used: As the mutations are introduced on purpose and systematically, their position in the unfolded state-space is known. The only thing that needs to be done is the check whether a mutation invokes a violation of the $tioco$-conformance relation in the sub-DAG beneath it, or not. Thus, the language inclusion check does not need to start at the root of the DAG, but should rather start at the mutation and explore only the sub-DAG beneath the mutated action. This reduces the investigated state space drastically. The updated process can be seen in Figure 2 (b).

In order to apply this approach, two steps need to be executed: First, as the unfolded automata might contain infeasible paths, a reachability-analysis is needed to check whether the mutated location or transition can actually be reached from the initial location. If the path from the initial state to the mutated location is in the form of a tree, i.e. every location along the path has exactly one incoming transition, the reachability is comparatively easy and only the guards and clock resets along the trace have to be checked for satisfiability. The constraints that are created on the clocks need to be stored, so they can be attached to the initial state of the tioco check. If there are several traces leading to the mutated state, there is not only one constraint, but one per different trace. As only one of them has to be satisfiable for the mutated state to be reachable, a disjunction of all these constraints is stored.

After finishing the reachability analysis, the DAG can be pruned, leaving only the subgraph below the mutation. Then the language inclusion check can be applied, with the mutated location as initial location, to see if the mutation propagates to a real fault. The only change to the classical check is that the clocks are not set to zero at the initial location, but are defined by the constraints calculated in the last step.

If a counterexample is found, the test-case generator merges the trace(s) calculated in the reachability check with the counter-example found by the language inclusion, to gain a time-adaptive test case from the initial state to the $tioco$-violation.

This check naturally allows to reach far higher depths in the $k$-bounded language inclusion of the DAGS than could be achieved otherwise, as the exponential growth of the complexity only starts after the mutation.

## V. Pruning the Inputs

Partial specifications are valid resources for test-case generation, as long as the partial models still conform to the complete specification. $Tioco$ allows the underspecification of inputs, thus by pruning inputs in the DAG, the $tioco$ conformance is not violated, while removing any observables would. There are several possibilities for the pruning approach:

- Prune according to a manually predefined *test purpose*.
- At each depth, pick a subset $N$ of all inputs, either randomly or according to some predefined distribution, and prune every input which is not contained in $N$.
- At each depth, only allow *exactly one* random controllable and prune the rest.

Note that these pruning options can already be applied during the determinization, thus the pruning does not only decrease the complexity of the test-case generation, but can already increase the efficiency of the unrolling and determinizing. In the following, we want to present these approaches in detail:

### A. Test purpose

Our test purposes we have in mind are defined as sets of inputs for each depth, so that at each step only the defined inputs are explored. The test purposed have to be defined by the test engineer, which requires some knowledge about the system, but ensures that the DAG only covers the parts relevant for the user. For the car alarm system presented in the introduction, a test engineer might want to avoid those parts of the DAG that start with alternating $locking$ and $unlocking$ of the doors. A well chosen test purpose to avoid this is $\{lock\}, \{close\}, \{\}, \{open, unlock\}$. This prunes the DAG to locking and closing the doors in the first two steps, and avoids any inputs in the third step (thus the empty set). Thus, in the third step only outputs are received, and the DAG only covers the branch that arms the alarm system. Opening the door in the fourth step will cause the alarms to start and unlocking it will trigger the transition for unarming the system, so both important branches of the DAG are covered.

To complete the test case with all outputs that are triggered immediately after the test purpose, the test purpose can be completed by adding some empty lists at its end. Thus, in the final steps no new input is triggered, but all outputs are still captured. Contrary, if the test purpose should only be used to prune the first few steps of the DAG, and the unfolding should be continued afterwards, it suffices to add sets with all inputs, until the desired depth is reached.

### B. Automated picking of inputs

Picking a set of inputs $N$ for each depth, either per randomization or according to some distribution, is very similar to the manual approach in the previous subsection. The main advantage is that no knowledge about the specification is needed and the partial model can be created purely automatically.

The random approach of selecting the enabled inputs needs the least effort, even though it needs to ensure in some way,

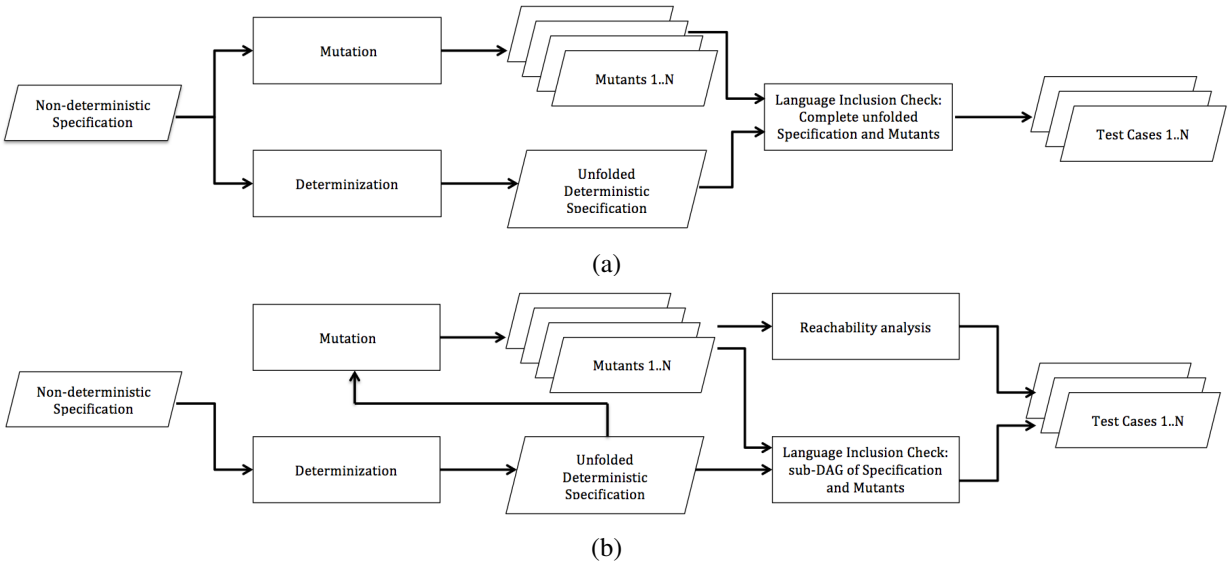


Fig. 2. Our test-case generation process: (a) illustrates the original process (b) shows the updated process, where the language inclusion is only applied to a sub-DAG of the specification and the mutants.

that the chosen inputs are actually enabled in some parts of the DAG in the current depth.

Choosing the inputs via statistical measures helps steering the partial model in the right direction. If locking and closing the doors has a higher priority then opening and unlocking, the probability for arming the system in the selected partial model is very high.

Alternatively, the inputs can also be chosen for each location in the DAG individually, instead of using the same inputs at each depth. This would allow specifying the priority of the inputs in the specification, dependent on the current location. Thus, for instance, the priority of $lock$ and $close$ might be higher in the first few locations, and might decrease as the inner parts of the model are explored.

### C. One input per depth

Choosing exactly one input per depth is a special case of the approach described in the previous subsection, where $|N|$ is set to one. Again, the selection can be done randomly or steered by heuristics. The partial models constructed by this approach are in fact already adaptive test cases: They provide one fixed sequence of inputs, and contain every branching caused by the outputs. Given a test driver that can handle adaptive tests, theses tests could immediately be executed on a system-under-test.

While these partial models can not be used for generating a test suite via model-based mutation testing, as they already are in the form of test cases, the model-mutations still can be used to assess the quality of these random tests, i.e. to check how many mutants are killed by the adaptive test cases. The mutation analysis could also be used as a stopping criterion for the test-case generation, indicating when enough test cases have been generated.

Note that we distinguish between two types of non-determinism: deterministic automata, where two transitions with the same label are enabled at the same time, and non-deterministic systems, where the system can chose non-deterministically between different outputs. While our previous approach was not able to handle the first case, the second one did not cause problems. However, our previous test cases only contained one specific trace through the system, and hence unexpected outputs led the test case to an $inconclusive$ verdict. The test cases we receive by pruning the tree are fully adaptive and contain every trace that can be invoked by the chosen inputs.

## VI. Related Work

Test-case generation from timed automata is not a new topic: The UPPAAL tool family contains a series of tools working with timed automata. Three of them are directly used for testing: UPPAAL Cover [5] generates tests offline and allows the specification of observers to generate tests satisfying pre-defined coverage criteria. Cover required the specification to be deterministic. UPPAAL Tron [6] is used for online testing, where inputs and delays are chosen non-deterministically and executed on the system-under-test and the specification simultaneously and all outputs that are received from the system are checked for conformance on the model. UPPAAL Yggdrasil is the newest testing tool in the UPPAAL family, but so far no publication about it is available. Recently, Wang et al. [9] published an approach for language inclusion from timed automata which may contain non-determinism. Similar to us, they unfold the automata, receiving an infinite tree. They use zone-abstraction and lower-upper bounds simulation to reduce the size of the tree before the language inclusion check. Finally, they use the language inclusion for the verification of timed patterns. Their approach does not consider silent transitons and is not intended for test-case generation. Bertrand et al. [4] published a paper about test-case generation from non-deterministic timed automata. They apply an approximate determinization via a game approach. Their determinization does not preserve the exact language of the timed automata, it only ensures $tioco$ conformance between the original and

the determinized model. Thus, their determinization approach avoids the state-space explosion we investigate in this paper. Aichernig and Jöbstl [1] proposed splitting their refinement check for action systems into a non-refinement check for each action and a reachability check, if non-refinement was detected. This contains similarities to the separated $tioco$ and reachability checks, we presented in Section IV. They applied their check on-the-fly, during the exploration of the state space fo their system, whereas we start our approach with a given tree covering the whole state space. As their experiments showed huge improvements in runtimes, we hope to be able to achieve similar results for the timed case.

## VII. Conclusion and Future Work

Within this paper we proposed two methods to select suitable partial models from timed DAGs, in the context of model-based testing. The restriction to partial model helps avoiding the exponential state-space explosion caused by unfolding timed automata and thus will allow us the effective language inclusion from unfolded and determinized timed automata. This enables the application of our previous test-case generation technique for deterministic timed automata to deterministic timed automata with silent transitions. The resulting test cases are adaptive, as they still contain all outputs, and deterministic, as they were built from the determinized system. In future steps we plan to evaluate both approaches on industrial case studies and investigate their scalability in the context of complex networks of timed automata.

## References


[1] Bernhard K. Aichernig and Elisabeth Jöbstl. Towards symbolic model-based mutation testing: Combining reachability and refinement checking. In Alexander K. Petrenko and Holger Schlingloff, editors, *Proceedings 7th Workshop on Model-Based Testing, MBT 2012, Tallinn, Estonia, 25 March 2012.*, volume 80 of *EPTCS*, pages 88–102, 2012.

[2] Bernhard K. Aichernig, Florian Lorber, and Dejan Ničković. Time for mutants - model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs*, volume 7942 of *LNCS*, pages 20–38. Springer Berlin Heidelberg, 2013.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] Nathalie Bertrand, Thierry Jron, Amlie Stainer, and Moez Krichen. Off-line test selection with test purposes for non-deterministic timed automata. In ParoshAziz Abdulla and K.RustanM. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 96–111. Springer Berlin Heidelberg, 2011.

[5] Anders Hessel and Paul Pettersson. Cover-a test-case generation tool for timed systems. *Testing of Software and Communicating Systems*, pages 31–34, 2007.

[6] Marius Mikucionis, Brian Nielsen, and Kim G. Larsen. Real-time system testing on-the-fly. In Kaisa Sere and Marina Waldén, editors, *the 15th Nordic Workshop on Programming Theory*, number 34 in B, pages 36–38, Turku, Finland, October 29–31 2003. Abo Akademi, Department of Computer Science, Finland. Abstracts.

[7] Amnon Rosenmann, Florian Lorber, Dejan Ničković, and Bernhard K. Aichernig. Bounded determinization of timed automata with silent transitions. Technical Report IST-MBT-2015-01, Graz University of Technology, Institute for Software Technology, 2015. Online. https://online.tugraz.at/tug_online/voe_main2.getVollText?pDocumentNr=1003322&pCurrPk=83975.

[8] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, pages 1–38, 2008.

[9] Ting Wang, Jun Sun, Yang Liu, Xinyu Wang, and Shanping Li. Are timed automata bad for a specification language? language inclusion checking for timed automata. In Erika brahm and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 310–325. Springer Berlin Heidelberg, 2014.