Conformance Testing in UPPAAL: A diabolic approach

E. J. Njor
Dept. of Computer Science
Aalborg University
Aalborg, Denmark
enjor16@student.aau.dk

F. Lorber
Dept. of Computer Science
Aalborg University
Aalborg, Denmark
florber@cs.aau.dk

N. I. Schmidt
Dept. of Computer Science
Aalborg University
Aalborg, Denmark
nschmi16@student.aau.dk

S. R. Petersen
Dept. of Computer Science
Aalborg University
Aalborg, Denmark
srpe16@student.aau.dk

***Abstract*—Model-based mutation testing is a fault-based method in the model-based testing area of research. It has been applied to several modeling formalisms, including timed automata. We propose a model transformation termed "diabolic completion" that allows for conformance testing directly in the UPPAAL tool. We have also developed a system to automate most of the process, which include taking a model, and performing diabolic completion, with the additions of allowing creation of mutants, conformance checking using the UPPAAL verification engine, and test case generation. We then set up a case study using a car alarm system model, which has been used several times in this area of research, and compare the efficiency with two existing tools, Ecdar 2.2 and MoMuT::TA, observing a significant speedup.**

I. Introduction

The importance of correct systems cannot be understated, especially in the case of safety-critical systems, such as for instance air traffic control systems. The current industry solution to deal with correctness is to thoroughly test the system. This testing is usually conducted by a tester, with a best effort "ad-hoc" approach, to the point where the tester believes the system to be correct. The tester, however, can unfortunately miss major bugs, which can have fatal consequences.

To get around having a manual tester, the concept of *model-based testing* has been proposed. In model-based testing, one creates a mathematical model of the system, called the specification, which can then be used to generate test cases for the *System Under Test* (SUT) [7]. Several approaches to this have been proposed, ranging from manual testing using the model, up to structural coverage tests [23]. In this paper we look into a fault-based approach called *Model-Based Mutation Testing* (MBMT) where a specification is mutated to create potentially faulty models, called *mutants*. Then the mutants are checked for conformance to the original model. If a mutant does not conform to the model, this shows that the mutant contains an observable fault and produces a counter example. Using this counter example, a test case can be generated for an SUT of the specification, based on the introduced fault. In current society, software systems are reaching new heights in complexity and importance. This results in the need for better and more practical testing methods, which MBMT tries to address. What makes MBMT an appropriate method for combating this, is the systematically testing of a SUT, giving better coverage of certain faults determined by the mutation operators, and the semi-automation of the testing process, which results in it being cheaper than manual or exhaustive testing techniques [17] [20].

One important aspect of many safety-critical systems is the need to adhere to real-time specific requirements. In such systems, it is not only important that a system needs to provide a specific output in certain situations, but also that the output is given at a specific point in time, or before some deadline expires. One formalism for specifying such specifications are timed automata [5], which are extended finite state machines that use variables called clocks to measure time.

Complete systems for creating mutations and generating test cases for timed automata are at the time of writing, to the writers knowledge, limited to Ecdar 2.2 and MoMuT::TA. In this paper we propose a method to do conformance testing and test case generation using the verification engine of UPPAAL [16], a widely used and very efficient model-checker for timed automata. We then do an evaluation of this method on a car alarm system, and compare the results with the two previous tools.

The remainder of the paper is structured as follows. First, in Section II we will provide preliminaries on timed automata and model-based mutation testing. Then, in Section III we discuss related work. Next, in Section IV, we will show how to apply model-based mutation testing using UPPAAL. In Section V and Section VI we will give details on our implementation and demonstrate results of our approach, comparing to previous techniques. Finally, in Section VII and Section VIII we will provide conclusions and an outlook on future work.

II. Preliminaries

A. UPPAAL Timed automata

Timed automata, originally described in Alur et al. [5], are an extension of finite state machines that include real-valued clocks that keep track of the time passed, which can be reset along transitions and used in constraints in locations and transitions. The time domain used for the clocks is $\mathbb{R}^+$.

The reasoning for introducing this extension at the time, was to be able to more accurately model physical processes compared to other possible models, such as: finite automata [21], non-deterministic Büchi automata [6], deterministic and

nondeterministic Muller automata [19], etc. These types of automata are all examples of methods that have abstracted away time. This can cause problems in time sensitive systems.

Definition 1. The set of possible guards $\Phi(C)$ over the set of clocks C is defined by the abstract syntax: $\delta ::= c \sim n | c - d \sim n | \delta \wedge \delta$, given $c, d \in C$, and $n \in \mathbb{N}$.[1]

Definition 2. The UPPAAL version of a timed automata, which is partially derived from Aceto et al. [1], is a tuple $A = (\Sigma, L, l_0, I, C, E)$, where:

- $\Sigma = \Sigma_I \cup \Sigma_O$ is a finite, non-empty set of symbols, either specified as input or output on the form $\alpha? \in \Sigma_I$ or $\alpha! \in \Sigma_O$ respectively, given α is in the language alphabet.
- L is a finite, non-empty set of locations.
- $l_0 \in L$ is the initial location.
- $I : L \to \Phi(C)$ assigns invariants to locations.
- C is a finite set of real valued clocks.
- $E \subseteq L \times L \times \Phi(C) \times \Sigma \times C$ is a finite set of transitions. A transition $e = (l, l', \delta, a, \lambda)$ is a transition from location l to l', given the guard δ is satisfied, with the input/output a, and clock resets λ.

Definition 3. A timed trace σ is a sequence of pairs of time delays and actions, denoted as $[\,(t_1, a_1), (t_2, a_2), ..., (t_k, a_k)\,] \mid t_i \in \mathbb{R}^+,\ a_i \in \Sigma$, for some positive integer i, given $i \leq k$. A timed trace σ can be applied to a model M, denoted by $M(\sigma)$, in which case for each $(t_i, a_i) \in \sigma$ an action of the timed trace is applied to the model after its respective time delay. At the end of a timed trace σ we end up in a state (l, C), denoted by $M(\sigma) = (l, C) | l \in L$, where C is a set of clocks and their values, and l is the location we end up in.

We only consider *deterministic* timed automata as input models to our approach, meaning that at no point in time will there be two transitions with the same label leaving a state. In addition we do not consider timed automata with silent transitions, i.e., transitions without an input or output label.

Example. To show an example of a timed UPPAAL automaton, we use the specification of a vending machine, shown in Figure 1. Its formal definition is:

- $\Sigma = \{btnc?, btnt?, coffee!, tea!\}$
- $L = \{S1, S2, S3\}$
- $l_0 = S1$
- $I(S1) = true$,
  $I(S2) = true$,
  $I(S3) = true$
- $C = \{x\}$
- $E = \{(S1, S2, x > 2, btnc?, \{x := 0\}),$
  $\{(S1, S3, x > 2, btnt?, \{x := 0\}),$
  $\{(S2, S1, x < 3, coffee!, \{\}),$
  $\{(S3, S1, x < 3, tea!, \{\})\}$

B. Model-Based Mutation Testing

Model-Based Mutation Testing (MBMT) is a subfield of the model-based testing techniques. The approach relies on an assumed to be correct specification, which is used to generate tests for a possibly malfunctioning SUT. MBMT utilizes a fault-based approach, where it generates new models using mutations based on the specification of the SUT. A mutant is an altered version of the specification it is based of, with one or more changes introduced. The changes are made with mutation operators, where an effectively infinite amount of operators are possible. Some of the operators for timed automata introduced in Aichernig et al. [3], are as follows:

- Change action: The action on a transition is changed.
- Change target: A transitions target location is changed.
- Change source: A transitions source location is changed.
- Change guard: The guard on a transition is changed.

The purpose of the mutation operators is to introduce common mistakes typically found in the SUTs.

After this, typically model completion tools are applied on the mutant and specification to make them input-enabled. Then, we check for conformance between each mutant and the specification. In real-time systems, the timed input-output conformance *tioco* [13] is generally used as the conformance relation. An implementation I conforms to a specification S with respect to tioco, iff for all traces through the specification, the outputs of I are a subset of the outputs of S, where the passage of time is considered an output as well. Thus, I must never show additional output behavior, or delay longer than S. Assuming that the mutant does not conform to the specification, i.e., it shows new incorrect output behaviour, the conformance check will produce a counter example, which is a trace through the mutant that is not possible in the specification. Finally, these traces can be transformed into test cases and executed on the SUT to see whether it conforms to the specification or to one of the mutants.

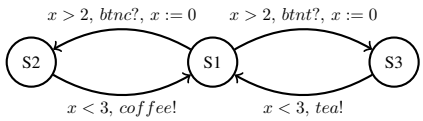


Fig. 1: Running example of a coffee vending machine.

III. Related Work

MBMT has already been applied to many types of models, including: probabilistic finite state machines [11], UML state machines [2] and Timed Automata, in MoMuT::TA [4], Ecdar 2.2 [9], [14] and, without an actual conformance check, in UPPAAL [18]. Aichernig et al. [4] were the first to apply model-based mutation testing to timed automata. They introduced the mutation operators and used the SMT-solver Z3 in their tool MoMuT::TA to perform a bounded tioco conformance check. Due to the boundedness of the approach, large systems with long traces would either not be completely explored, or need more time for the conformance check.

Larsen et al. [14] used the mutants created by MoMuT::TA to perform refinement checks with the tool Ecdar, producing adaptive test strategies instead of linear test cases. Later,

[1]UPPAAL allows for variables, which are treated as natural numbers.

Gundersen et al. [9], [14] developed and used the tool Ecdar 2.2, which supports mutation, test-case generation and test-case execution from within the tool.

Lorber et al. [18] proposed to use the properties of a specification to test whether mutants still fulfill these properties, instead of performing a complete conformance check. This reduces the computation time required and generates less test cases than the usual approaches, which are faster to execute.

Godskesen et al. [8] proposed a fault-based method using a connectivity fault model. They also use a kill state (in a seperate observer automaton) to identify the killing of mutants, and reachability queries to get the trace. Contrary to our approach, they test for errors that lie in the communication between the SUT and the environment, instead of in the SUT.

Krenn et al. [12] elevated the MBMT approach, by showing how to expand a trace gotten from a single mutated timed automaton into a trace of a network of timed automata, enabling the fault-based verification of composed systems.

General testing approaches using timed automata include the following: Larsen et al. [15] proposed to conduct conformance testing using online black box testing between a specification and an implementation, and Hessel et al. [10] conducted offline testing using an observer automata and an observer language to specify coverage criteria. This lead to the creation of UPPAAL TRON and UPPAAL CoVer, respectively.

The proposed approach aims to use the UPPAAL verification engine to achieve testing for conformance between mutants and their specification, which means we will, like Ecdar 2.2, achieve unbounded conformance testing, as opposed to MoMuT::TA that is bounded. The approach will provide more generality, as complete UPPAAL models can be supported, while previous approaches excluded certain elements such as C-like code or urgent/committed locations that are supported by UPPAAL, and, using the highly optimized UPPAAL eninge, be faster than previous approaches.

IV. Model-Based Mutation Testing in UPPAAL

In this section, we introduce our approach to perform MBMT directly in UPPAAL. The main idea behind this is to create the mutants and the specification as separate communicating timed automata, and have UPPAAL explore them in parallel. For that, the specification needs to undergo a transformation, which we named "diabolic completion". It works by adding a new location, which from this point forward we will refer to as the "kill state". The goal is to create the state and the transitions leading there in a way, such that it can only be reached if the mutant performs an output that cannot be taken by the specification. This reduces the conformance check to a reachability check for that location. The next subsection will expand on the different steps needed to utilize this transformation in the complete test-case generation method.

The workflow consists of the following steps:

- Create mutants
- Make the specification input enabled
- For each mutant
  - Make the mutant input enabled
  - Perform diabolic completion
  - Perform the reachability analysis
  - Generate test cases

A. Mutation.

The creation of the mutants follows the mutation operators proposed by Aichernig et al. [3]. However, even though the theory supports the full set of operators, as of now we only implemented the operators “change action”, “change source” and “change target”. We are only using first-order mutants, that is, each mutant only contains one introduced fault. An example of a ”change action” mutation can be seen in Figure 2a.

B. Making models input enabled.

In the second step, model completion is performed to make the models ready for conformance checking. Specifically demonic and angelic completion (see **Defintion 6-7**) are applied to the specification and mutants, respectively, to make them input-enabled. The idea behind demonic completion on the specification is to support underspecification, i.e., the implementation is allowed to react to more inputs than initially specified, but such an input would leave the specified area and from then on all behaviour is valid. The angelic completion of the mutants ensures that the mutants cannot block any input, but simply neglect any input that is not specified in their model. We need four definitions to formally define the completions.

Definition 4. We define a function we call the inverse location input function $f_{\neg in} : X \to Y$ that maps a location to a set of inputs, containing all the inputs not specified in any of the transitions that start from the given location. The domain of X is therefore L, and Y is $\mathcal{P}(\Sigma_I)$. The concrete function maps $x \to I'$, which is done given $I' = \Sigma_I - I$, and where $I = \bigcup_{e \in E | p_1(e)=x,\ p_4(e) \in \Sigma_I} p_4(e).\ p_i(e)$ is used to denote the i-th projection of the tuple e, that is, in this case the label of the edge.

Definition 5. We define the inverse guard function $f_{\neg guard} : X \to Y$ that maps a guard to the negated version of the guard , where the domains of X and Y is $\Phi(C)$. The concrete function maps $x \to A'$, given x is built using the abstract syntax in **Definition 1**. A' is defined as $A' = l'_1 \vee l'_2 \vee ... \vee l'_k$, for all $i \leq k$, where k is the number of simple constraints l in x[2]. For an l where $l = a \sim b$, for some natural number b, and where a is built using the abstract syntax $a ::= c | c - d$, l'_i is built in one of 5 ways:

- $l'_i = \{a \leq b\} | l_i = \{a > b\}$
- $l'_i = \{a < b\} | l_i = \{a \geq b\}$
- $l'_i = \{a > b\} | l_i = \{a \leq b\}$
- $l'_i = \{a \geq b\} | l_i = \{a < b\}$
- $l'_i = \{a < b \vee a > b\} | l_i = \{a = b\}$

Definition 6. We define a function we call the action resolve function $f_{resolve} : X \to Y$ that maps a tuple $x = \{E, l, l', a\}$,

[2]Note that guards in timed automata do not allow for disjunction. The implementation will split these transitions into multiple transitions.

which contains a set of transitions, a start location, an end location and an action, to a new set of transitions, containing all the transitions from E and a new one with action a, going from the start location l to the end location l', with a guard that ensures that it can only be taken, as long as the invariant allows it, and no other transitions with the same action and start location have a guard that allows traversal at the same time. The concrete function maps $x \rightarrow E'$, where $E' = E \cup e_{new}$, given $E_{act} \neq \emptyset$, where $E_{act} = \bigcup_{e \in E | p_4(e) = a} e$. The new transition $e_{new} = (l, l', g, a, \emptyset)$, where the guard $g = f_{\neg guard}(e_1) \wedge f_{\neg guard}(e_2) \wedge ... \wedge f_{\neg guard}(e_k) \wedge f_{\neg guard}(I(l))$, $\forall e_i \in E_{act}$.

1) Angelic completion: Angelic completion [22] is when a model is made input-enabled by creating self-loops on every location, making them partially reflexive, allowing them to take all possible inputs in every possible location.

Angelic completion is applied to the mutants, as real time systems are not considered to block unexpected input, but rather to ignore input if given at an unexpected time. An example of the angelic completion can be seen in Figure 2b. In this example shows that it would not make sense to change state if an unexpected input is given while it is making coffee, but rather continue making coffee until it is done.

Definition 7. Now we define angelic completion based on the previously defined functions, where we start by adding transitions to each location for inputs that are generally available in the location but disabled at certain times, $E' = E \cup f_{resolve}(E, p_1(e), p_1(e), p_4(e))$, which is done $\forall e \in E$. Then we add for each location the remaining transitions for inputs that are not enabled at all, $E'' = E' \cup (l, l, \emptyset, a_{in}, \emptyset)$, which is done for all $a_{in} \in f_{\neg in}(l)$, for all $l \in L$.

2) Demonic completion: Demonic completion [22], like angelic, concerns itself with making a model input-enabled. Demonic completion achieves this by creating a sink location, with transitions leading there from every location for every input unspecified for that location. I.e., these transitions are taken if we observe an unexpected input, which makes us leave the specified behaviour. The sink location itself accepts all input or output, making it completely reflexive.

Demonic completion is applied on the specification itself, because it should allow for implementations to expand on the features of the specification. It is a very real possibility that only a small part of a system is modelled (e.g., the safety-critical parts are modelled) and the implementation contains more functionality than the specification. In the case that an unspecified input is given, the behavior is unpredictable, but should still be accepted. An example of demonic completion can be found in Figure 2c. Should a new functionality be added (such an additional button to create hot chocolate), then that intuitively shouldn't fail the specification, which did not specify what happens if hot chocolate is requested.

Definition 8. Now we define demonic completion, on the previous defined functions, where we start with adding a new sink location, i.e., $L' = L \cup l_{sink}$. We then make that location both input and output enabled, by adding the transition $E' = E \cup (l_{sink}, l_{sink}, \emptyset, \Sigma_I \cup \Sigma_O, \emptyset)$. Now we will, like in angelic completion, add all transitions for guarded input transitions in each location $E'' = E' \cup f_{resolve}(E', p_1(e), l_{sink}, p_4(e))$, which is done $\forall e \in E$. We then add the remaining transitions $E''' = E'' \cup (s, l_{sink}, \emptyset, a_{in}, \emptyset)$, which is done for all $a_I \in f_{\neg in}(l)$ and $\forall l \in L$.

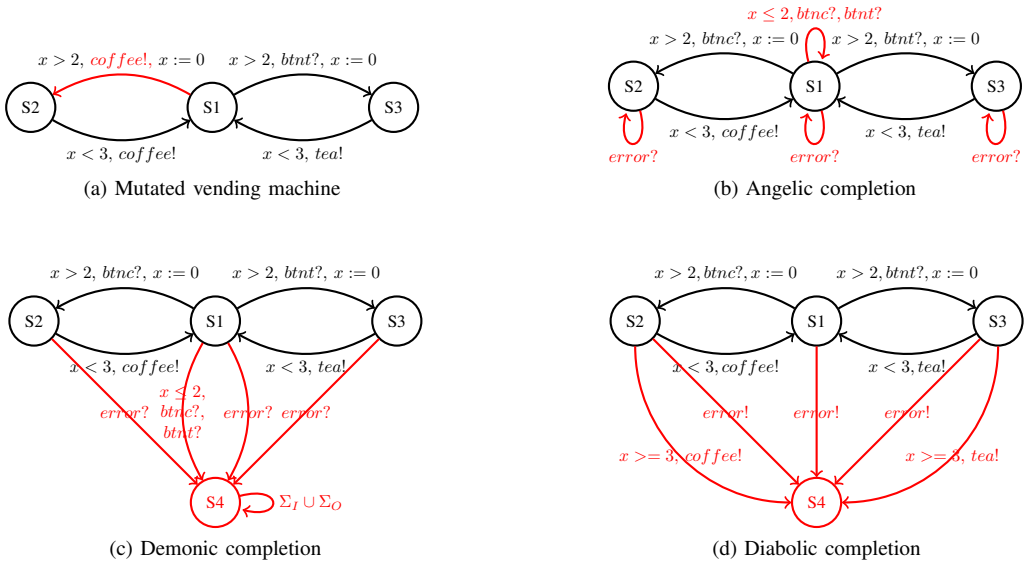
C. Diabolic Completion

Diabolic completion works by adding a new location, the "kill state". Once this state is reached, the mutant has been killed, meaning the mutant does not conform to the specification. The state is reached if the mutant gives any unexpected output (either an output not available in the current location, or an output that comes at a wrong point of time), which does not match an already existing transition of the current location in the specification. An example of diabolic completion can be found in Figure 2d. The transition with the action *error!* symbolizes every output that is not accepted in the source location, and, for instance, the transition with the action *coffee!* from $S2$ to $S4$ and the guard $x \geq 3$ covers the case where an output arises while the guard of the correct transition was not satisfied. After applying this transformation and creating the kill state, we can perform a reachability check on the parallel product of the specification and the mutant to find a case where the mutant can perform an output that leads the specification to the kill state.

1) Algorithm and mathematical definition: The algorithm for diabolic completion is shown in listing 1.

Listing 1: Algorithm for diabolic completion.

```
INPUT: A tuple M = (Σ, L, l0, I, C, E),
representing a UPPAAL model.
OUTPUT: A tuple H = (Σ, L, l0, I, C, E),
representing a UPPAAL model
after diabolic completion.
Diabolic_Completion(M){
  tuple H = M
/*Adding kill state*/
  H.S = H.S ∪ kill_state
/*Transitions added from each s to kill_s*/
  foreach l in M.L{
    set temp_actions = ΣO
    foreach e in M.E{
      if(e.l == l && e.a is an output){
        if(e.δ != NULL && e.a ∈ temp_action){
          H.E = resolve(H.E, e.l, kill_s, e.a)
        }
        temp_actions −= e.a
      }
    }
    foreach a in temp_actions{
      H.E = H.E ∪ (l, kill_s, NULL, a, NULL)
    }
  }
  return H
}
```



(a) Mutated vending machine

(b) Angelic completion

(c) Demonic completion

(d) Diabolic completion

Fig. 2: UPPAAL timed automata that show a mutation and angelic, demonic and diabolic completion of the vending machine. The red color indicates the changed parts. In the timed automaton, the $error?$ labels are used to show a set of inputs. The set $error? = \Sigma_I - p_3(e) | \forall e \in E \wedge p_1(e) = p_1(e_{error?})$, and, $error! = \Sigma_O - p_3(e) | \forall e \in E \wedge p_1(e) = p_1(e_{error!})$

The algorithm starts of by taking a tuple representing the model as input (see section II-A), where it first adds the new kill state seen in Line 9. After that, it traverses all locations, and for each location it creates a set of all output actions (Line 12) to check which ones were processed already. Then, for each outgoing transition of the current location that is an output (Line 14) and has not yet been processed, the *resolve()* function (Line 16) is applied. This function is defined mathematically in **Definition 6**, and creates a transition to the kill state with a negation of the guard of all transitions with the current source location and action. Next the algorithm removes the current action from the set of actions that still needs to be processed (Line 18). Finally, for each action left in the temporary variable, a transition is created from the current location to the kill state, with the given action and an empty guard, seen in Line 22.

Definition 10. We define a function we call the inverse location output function $f_{\neg out} : X \rightarrow Y$ that maps a location to a set of outputs, containing all the outputs not used in any of the transitions starting from the given location. The domain of X is therefore L, and the domain of Y is $\mathcal{P}(\Sigma_O)$. The concrete function is $x \rightarrow O'$, which is done given $O' = \Sigma_O - O$, and where $O = \bigcup_{e \in E | p_1(e)=x,\ p_4(e) \in \Sigma_O} p_4(e)$.

Definition 11. We define the process of diabolic completion, using the functions previously defined, by adding a kill state, i.e., $L' = L \cup kill$. Then we add the output transitions for outputs that are generally available in a location but disabled at certain times, $E' = E \cup f_{resolve}(E, p_1(e), kill, p_4(e))$, which is done for all $e \in E$. And finally we add all the remaining transitions for outputs that are not enabled at all, $E'' = E \cup (l, kill, \emptyset, a_{out}, \emptyset)$, which is done for all $a_{out} \in f_{\neg out}(l)$, which in turn is also done for all $l \in L$.

D. Detecting non-conformance

After applying all the described transformations, detecting non-conformance basically is reduced to a reachability-check on the kill state. To do that, the specification and a mutant need to be explored in parallel, so that each action one takes is also taken on the other model. Thus, the reachability check on the kill state in the specification is performed on the parallel product of the the specification and a mutant. Assume a specification S, that was made input-enabled by demonic completion and a mutant M, turned input enabled via angelic completion. By applying diabolic completion to the specification we received S'. Then we can build a network of communicating timed automata, consisting of the specification after demonic and diabolic completion S', and the mutant that has undergone angelic completion M. If the kill location is reachable in that network, this shows that the mutant did not conform to the specification, and the other way around.

Theorem 1: $\forall \sigma \in M : S'(\sigma) = (kill, C) \leftrightarrow \sigma \in M \wedge \sigma \notin S$, where C is a set of clock valuations.

Proof 1: We intend to show that the left side of the statement is equivalent to the right side, and therefore have to split the proof into two parts:

1) $\forall \sigma \in M : S'(\sigma) = (kill, C) \rightarrow \sigma \in M \wedge \sigma \notin S$
2) $\forall \sigma \in M : \sigma \in M \wedge \sigma \notin S \rightarrow S'(\sigma) = (kill, C)$

We intend to prove the 1) part of the proof first, and will do so using proof by contradiction. Therefore our new negated assumption would be:
$\exists \sigma \in M : S'(\sigma) = (kill, C) \wedge \sigma \notin M \vee \sigma \in S$
This we can reduce, as $\sigma \notin M$ will never be true, considering

that we are looking for $\exists\sigma \in M$. The reduced expression would be: $\exists\sigma \in M : S'(\sigma) = (kill, C) \wedge \sigma \in S$

Remember S' is the model S, but where S' had diabolic completion performed on it, which added a new location $kill$, and transitions to reach it. This means that there cannot exist a trace where both $S'(\sigma) = (kill, C)$, and $\sigma \in S$ is true, since the step from location l' that leads to $kill$ would be an output action not accepted by any transitions in the corresponding location l in S. This is therefore a contradiction, proving 1) to be a true statement.

We now intend to prove the 2) part of the proof, and will again do so by contradiction. As such our new negated assumption would be: $\exists\sigma \in M : \sigma \in M \wedge \sigma \notin S \wedge S'(\sigma) = (kill, C)$. Again, we eliminate $\sigma \in M$, and receive: $\exists\sigma \in M : \sigma \notin S \wedge S'(\sigma) = (kill, C)$

Remember S is input-enabled, meaning it at all times can take any input, and should it reach the state $(sink, C)$, then any input and output would be accepted. Therefore, for a $\sigma \notin S$, we would need some output step $(t_i, a_i) \in \sigma : p_2((t_i, a_i)) \in \Sigma_O$, so that $\sigma \notin S$. Also remember that diabolic completion, makes a model output-enabled, meaning that every output in σ either was present in S or leads to the kill state. Thus, such a step (t_i, a_i) cannot exist. This is therefore a contradiction, proving 2) to be a true statement, finishing our proof.

E. Generating Test Cases

The conformance testing of the previous section can either result in showing non-conformance or conformance between the specification and the mutant. In the case that non-conformance is shown, we say that the mutant has been *killed*. This also means that the mutant has introduced a fault into the system that is being modeled. The trace that discovered the fault in the mutant can be extracted from UPPAAL, and a test case can be generated for the SUT, that tests the actions that leads the mutant to its fault. Thus we have created a test that shows whether or not the fault introduced by the mutant is present in the SUT.

V. Implementation

We have created an implementation for automatic diabolic completion of a model provided as input, and with the possibility of going through the whole MBMT process of generating mutants, checking conformance with the specification, and test case generation based on the mutants. This was accomplished employing our transformation on the specification, and interfacing with UPPAAL to make use of their verification engine. This was done using the library written in Java, made publicly available by the creators of UPPAAL. In the following sections, we will go into detail with how the theory is linked to the implementation.

A. From theory to implementation

In **Definition 6.** the function $f_{resolve}$ is defined, which creates a new transition to the kill state, based on each outgoing transition with the same action as the one given to it. The guard created for the new transition is combined based on all the inverted guards of these transitions with the same action. Since UPPAAL does not allow for the use of disjunction in its guards, a problem occurs when conjunctions are negated. For the case of the guard only containing one or more disjunctions, we simply split the guard into multiple edges, whereas if the guard contain both one or more conjunctions and disjunctions combined, it gets a bit more complicated. The way we avoid this problem, in our system, is by the transformation we will refer to as the "chain transformation". An example of the resulting model after applying chain transformation on Figure 3a can be seen in Figure 3b. What happens is that a location is created for each conjunction in the guard, and for each of the sides of a disjunction a transition is created to reach the location / leave it towards the target location. The way it should be understood is that each added location represents a single conjunction, and the transitions between the locations represents the logical expressions on all sides of disjunctions between two conjunctions. Such a transformation does not result in an equivalent automaton. However, it preserves reachability of the location behind the transformation, which in our case is the sink location.

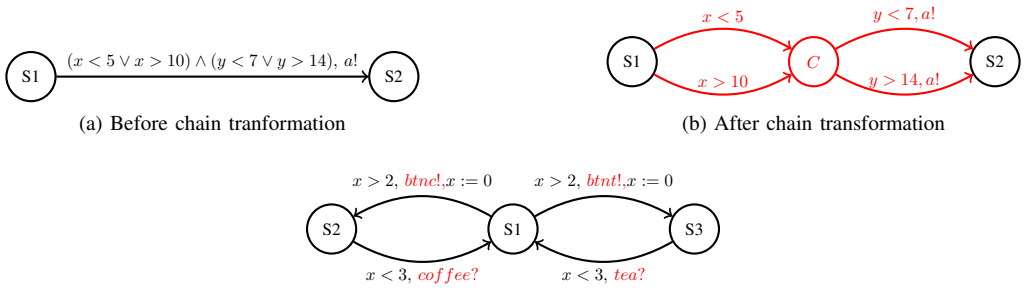
In UPPAAL it is possible to create a network of models that communicate with each other using the defined input and output channels. Since we make use of the UPPAAL verification engine to test if a mutant conforms to a specification, we have to make sure the mutant and specification can communicate. The way we achieve this, is by transforming all of the mutants by flipping all the input channels to output channels, and all the output channels to input channels. The resulting set of transitions after this transformation is: $E' = (E - e) \cup (p_1(e), p_2(e), p_3(e), f_{io}(p_4(e)), p_5(e)) \forall e \in E$, where the function f_{io} used is a function that maps from $X \rightarrow Y$, where $X, Y \subseteq \Sigma_I \cup \Sigma_O$. The concrete function maps $x \rightarrow \alpha!$, which is done for all x, where $x = \alpha?$. It also maps $x \rightarrow \alpha?$, which is done for all x, where $x = \alpha!$. An example of this transformation can be seen done on figure 1, and the result in figure 3c.

B. System execution

First, the mutants are generated, where every possible mutant is created based on the mutation operators: "change source", "change target", and "change action to output action".

Then some preprocessing is done on the specification to set up for the other stages. Specifically the system goes through every location of the model, where for each location that has an invariant, the invariant is removed and combined with every guard of the outgoing edges, that is, if we have a guard g and an invariant i, the resulting guard g' would be $g' = g \wedge i$.

Then, angelic completion is applied on all mutants, and demonic- and diabolic-completion is done on the specification. At this point it is possible that the specification model has transitions containing disjunctions that UPPAAL does not allow, and, as mentioned, we use the chain transformation explained in section V-A, to remove them from the specification. Finally, we flip the outputs of each mutant, to allow for the possibility of communication between the mutant and specification.



(a) Before chain tranformation

(b) After chain transformation

(c) Flip of input/output

Fig. 3: Timed UPPAAL automata showing an example of a transition containing both conjunction and disjunction and the results of applying the chain transformation on it. It also show an example of flipping the output to input, and input to output, based on the vending machine model shown in figure 1. Everything colored red are to be understood as the changes to the model, and C inside a location as it being commited (no time may pass while in this location).

At this point in the execution, the conformance testing of the mutants is possible using the UPPAAL verification engine. This is done by checking for whether it is possible for the specification to reach the kill state, using reachability queries. This is done by making the query *E<> Spec.KillState*, where "Spec" is the template, using the integrated query language made for verification in UPPAAL. Assuming reaching the killstate was possible, UPPAAL outputs the shortest trace to reaching the killstate.

C. Test-case generation

UPPAAL contains the test-case generation feature Yggdrasil. This feature allows to add test code to transitions and locations. This test code can be of any form, but one simple way to do it is to call functions in input transitions, and check via assert statements for output transitions. For the coffee machine example that might be "ButtonPressed();" for the button transition, and "assert(coffee);" for the coffee output transitions. In addition, one can specify prefix and postfix code to ensure that the file contains everything you need to run the test case. The trace provided by UPPAAL is then used to generate test cases for the SUT, by combining all the test code of the transitions traversed in the trace, where the test code should be written into the transitions in the initial model. The system creates a file for each of the test cases generated, which can then be imported as tests into the SUT.

However, this test code only contains the positive transitions, and no code for the final transition that leads to the killstate. If one sticks to the pattern of only using assert statements in the output transitions test code, one could simply add a negation for that step. However, since we wanted to keep our program general, this was not done in our program. This means that the test code for the final transition needs to be manually added, while we only add the label of the transition in the file to indicate which transition led to the sinkstate.

VI. Case Study

We conducted a case study, analysing the the execution time for the entire execution process of creating mutants, doing conformance testing, and the test output. This was done on Ecdar 2.2, MoMuT::TA and our system, using the same car alarm system previously used in Aichernig et al. [4] and Gundersen et al. [9].

Program	Time
Our Program	00:02:43
Ecdar 2.2	00:03:16
MoMuT::TA	00:43:45

TABLE I: Runtime for Ecdar 2.2, MoMuT::TA and our system.

A. Test setup

The test was setup on a computer with the operating system Windows 10 Enterprise, a Intel Core i7-5600U CPU and 8GB of DDR3 RAM. We limited the operators in both tools to: “change source”, “change target”, and “change action to output action”.

The output our system gives is in a plain text format, containing generated unit tests as explained in section V, whereas MoMuT::TA gives a trace showing the non-conformance, which the user has to parse to use.

B. Results

Using the setup outlined in the earlier, we ran both systems on the car alarm system. The results are shown in table I.

It is clear that both Ecdar and our tool were significantly faster than MoMuT::TA on the given case study. However, the case study does not contain any of the elements where the SMT-solving used by MoMuT::TA has an advantage. In any model using data variables or parameters, MoMuT::TA might compare a lot better. Hence, before testing several different studies, these results are not conclusive. In addition, MoMuT::TA has, at the current time, more mutation operators implemented, and provides limited support for non-determinism. We were also faster than Ecdar, though to a way lesser degree. The speedup gained was most likely based on the fact that we only need to perform a reachability check instead of the full conformance check.

VII. Conclusion

We have proposed a method for achieving conformance testing directly in UPPAAL. This has been made possible using the model transformation we have termed "diabolic completion", and using the already optimized verification engine in UPPAAL to determine whether or not there is conformance between a specification and its mutant(s).

We have made an implementation in Java, using the library made available by the creators of UPPAAL. The implementation is able to automatically do the model transformation of diabolic completion, based on a model written in the same format as the ones used in UPPAAL. It is also able to do mutant creation using the mutation operators we have implemented: Change source, change target, and change action. The final functionality implemented is the automatic test case generation, collecting the test code along the witness trace using the Yggdrasil feature of UPPAAL.

We ended up comparing the performance of the mutation generation and conformance testing between Ecdar 2.2, MoMuT::TA and our system. This was all done using the same mutation operators, to ensure a fair comparison. The results show a significant speedup.

In addition, our approach provides more generality, as all UPPAAL features are allowed in the model. However, the mutation operators currently do not focus on these features.

VIII. Future Work

One limitation of our implementation is that it can only take a single model, also known as a template in UPPAAL, as input. This has the implication that integrated systems cannot be tested in our system. Therefore being able to do this, would open up for more complex systems to be tested using our implementation

Currently our system only supports single order mutants with one fault per mutant. There are both advantages and disadvantages to single order compared to higher order mutants.. The implementation of the option to use higher order mutants would increase the flexibility of our system.

We also believe there would be value in exploring new mutation operators affecting all of UPPAALs features, and conducting an empirical evaluation of mutation operators, to determine which ones would be best at discovering faults in general, or specific types of faults, in the SUT. We believe this could increase the flexibility of MBMT as a whole, and help users understand what mutation operators to choose for certain types of systems they want tested.

References

[1] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive systems: modelling, specification and verification*. cambridge university press, 2007.

[2] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. *Software Testing, Verification and Reliability*, 25(8):716–748, 2015.

[3] 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*, pages 20–38, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] Bernhard K. Aichernig, Florian Lorber, and Dejan Ničković. Time for mutants — model-based mutation testing with timed automata. In *Tests and Proofs*, pages 20–38. Springer Berlin Heidelberg, 2013.

[5] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[6] J. Richard Büchi. *On a Decision Method in Restricted Second Order Arithmetic*, pages 425–435. Springer New York, New York, NY, 1990.

[7] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 285–294, May 1999.

[8] Jens Chr. Godskesen, Brian Nielsen, and Arne Skou. Connectivity testing through model-checking. In David de Frutos-Escrig and Manuel Núñez, editors, *Formal Techniques for Networked and Distributed Systems – FORTE 2004*, pages 167–184, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[9] Tobias R. Gundersen, Florian Lorber, Ulrik Nyman, and Christian Ovesen. Effortless fault localisation: Conformance testing of real-time systems in ecdar. In Andrea Orlandini and Martin Zimmermann, editors, Proceedings Ninth International Symposium on *Games, Automata, Logics, and Formal Verification,* Saarbrücken, Germany, 26-28th September 2018, volume 277 of *Electronic Proceedings in Theoretical Computer Science*, pages 147–160. Open Publishing Association, 2018.

[10] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. *Testing Real-Time Systems Using UPPAAL*, pages 77–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[11] R. M. Hierons and M. G. Merayo. Mutation testing from probabilistic finite state machines. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 141–150, Sep. 2007.

[12] Willibald Krenn, Dejan Nickovic, and Loredana Tec. Incremental language inclusion checking for networks of timed automata. In Víctor A. Braberman and Laurent Fribourg, editors, *Formal Modeling and Analysis of Timed Systems - 11th International Conference, FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proceedings*, volume 8053 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2013.

[13] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, Jun 2009.

[14] K. G. Larsen, F. Lorber, B. Nielsen, and U. M. Nyman. Mutation-based test-case generation with ecdar. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 319–328, March 2017.

[15] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing*, pages 79–94, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[16] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. J. Softw. Tools Technol. Transf.*, 1(1-2):134–152, December 1997.

[17] F. Lorber, K. G. Larsen, and B. Nielsen. Model-based mutation testing of real-time systems via model checking. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 59–68, April 2018.

[18] F. Lorber, K. G. Larsen, and B. Nielsen. Model-based mutation testing of real-time systems via model checking. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 59–68, April 2018.

[19] David E. Muller. Infinite sequences and finite machines. In *Proceedings of the 1963 Proceedings of the Fourth Annual Symposium on Switching Circuit Theory and Logical Design*, SWCT '63, pages 3–16, Washington, DC, USA, 1963. IEEE Computer Society.

[20] M. Papadakis and N. Malevris. An empirical evaluation of the first and second order mutation testing strategies. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 90–99, April 2010.

[21] Michael Sipser. Introduction to the theory of computation. *SIGACT News*, 27(1):29–37, March 1996.

[22] Jan Tretmans. *Model Based Testing with Labelled Transition Systems*, pages 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[23] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.