MoMuT::UML
Model-based Mutation Testing for UML

Bernhard Aichernig†, Harald Brandl†, Elisabeth Jöbstl‡, Willibald Krenn*, Rupert Schlick* and Stefan Tiran*†

*AIT Austrian Institute of Technology, †Graz University of Technology, ‡AVL List GmbH
{name.surname}@ait.ac.at, {name.surname}@tugraz.at, {name.surname}@avl.com

Abstract—Model-based mutation testing (MBMT) is a promising testing methodology that relies on a model of the system under test (SUT) to create test cases. Hence, MBMT is a so-called black-box testing approach. It also is fault based, as it creates test cases that are guaranteed to reveal certain faults. MBMT is computationally expensive. The tool MoMuT::UML is the result of a multi-year research effort to bring MBMT from the academic drawing board to industrial use. In this paper we present the current stable version, share the lessons learnt when applying two generations of MoMuT::UML in an industrial setting, and give an outlook on the upcoming, third, generation.

I. INTRODUCTION

Fault-based testing plays an important role in the verification and validation of systems as it is able to demonstrate the absence of certain faults [1]. Model-based mutation testing (MBMT) is a particular instance of fault-based testing. MBMT is working in a black-box setting, which means that it does not look at the implementation during test case generation. Instead, it is working off a model of the system under test (SUT) that is directly derived from the requirements and usually is more abstract than the implementation. Hence, MBMT is able to generate test cases for systems lacking an accessible implementation (e.g., integrated circuits).

Given a behavioural model of the SUT and a set of generic fault models, i.e. so-called mutation-operators, MBMT strives to automatically generate test cases that are able to reveal whether any modelled fault has been implemented. To this end, MBMT will take the original model, apply one mutation operator at one particular location a time, deriving a so called mutant, compare the behaviour of the original model with the one of the mutant, and - once a difference is found - write out a test case that steers the SUT towards this difference. As a result, MBMT achieves a fault coverage of the model and the resulting test suite will uncover whether any of the modelled faults was implemented. In connection with the coupling effect, MBMT is able and has shown to produce high quality test suites (cf. [2], [3]) that will also satisfy certain control and data flow coverage criteria [4, page 186].

The principle drawback of MBMT is the computational cost associated to comparing the behaviours of the original model and the mutant. This amounts to a model-checking problem and, hence, typically suffers from the same issues, like state space explosion and the support of a restricted set of mathematical functions only. In difference to white-box testing or model-checking of source code, however, MBMT has the advantage of working with an abstract, simplified model of the SUT. Hence the setting is more amenable to automated reasoning and the elements of the modelling language exposed to the user can be well balanced to keep the computational costs within bounds. To further reduce costs, partial models can be employed (cf. [5]) and incremental techniques used (cf. [6]). Another challenge MBMT has to deal with is that some mutations do not lead to a different behaviour of the mutant. These so called equivalent mutants do not yield any test case, since they cannot be detected. However, they still consume computation time and contribute to the overall cost of MBMT. Finally, the selection of fault models also influences computational cost in another way: if the fault models are not chosen well a lot of duplicate tests might be computed, as different faults can yield the same test case.

The tool MoMuT::UML (pronounced ”MoMuT for UML”) is an attempt to investigate and overcome the challenges posed. The name stems from MoMuT’s main test case generation strategy, namely MOdel-based MUtation Testing. Since its inception in the MOGENTES project, the tool has been evaluated in a number of industry-driven case studies and research projects (cf. [2], [5], [7]), with yet unpublished work ongoing. The current stable version of MoMuT::UML supports two different test-case generation back ends, different test-case generation strategies and different conformance relations. MoMuT needs a conformance relation to decide whether the behaviour of a mutant is considered different from the original model. This paper describes the stable version of MoMuT::UML, the lessons learnt while evaluating the tool, and hints at improvements under development for the next major version.

The paper is structured as follows. Section II presents an overview of MoMuT::UML’s main workflow. This is followed in Section III by a detailed description of the UML models the tool is able to work with. Then, in Sections IV and V the current test case generation back ends are presented before Section VI discusses the lessons learnt of applying MoMuT::UML. Before the paper concludes in Section VIII a brief look at related research is given in Section VII.

†https://www.momut.org
‡http://www.mogentes.eu
§A non-commercial evaluation version is available at www.momut.org
II. Overview: UML-based Mutation Testing With MoMuT::UML

MoMuT::UML uses UML state charts, class diagrams, and instance diagrams as input for the test case generation. In addition a small UML-profile is employed so the tool knows which events, e.g., method calls, are supposed to come from the test environment and which are provided by the SUT in response. In our terminology the former ones are called ”controllables” (ctr), while we refer to the latter ones as ”observables” (obs). Notice that UML state charts form an asynchronous model, which means that controllables (input to the SUT) and observables (output of the SUT) can be interleaved in an almost arbitrary way. While this leads to more expressive models, it also makes test case generation more challenging and is unlike in synchronous systems where a controllable is always followed by an observable ad infinitum.

Figure 1 shows the principle architecture of the tool. Because the execution semantics of UML state charts is not fully defined MoMuT::UML needs to settle for one interpretation and converts the UML model to an object oriented action system (OOAS), as explained in [8], [9]. OOASs are a simple extension to action systems as proposed by Back et al. [10]. They model parallel processes through nondeterministic choice of actions and their formal semantics are defined using the weakest precondition predicate transformer [11]. MoMuT::UML will also prepare mutated versions of the UML model by applying a set of user-selectable mutation operators to a set of user-selected name-spaces. The resulting UML-model mutants are converted to OOAS as well. After the OOAS of the original model and all mutants are available, MoMuT::UML is ready for test case generation.

MoMuT::UML offers two distinct test case generation backends. One back end, developed over the course of the MOGENTES project, runs an explicit state exploration of the system, while the other one, developed within the TRUFAL project, uses symbolic techniques. In the following we refer to the former as ”enumerative back end”, while we call the latter ”symbolic back end”. Both back ends support three different test case generation strategies: (1) random, (2) mutation, and (3) combined random+mutation. All the strategies have in common that - conceptually - they work on a labelled transition system (LTS, cf. [8]) representation of the given OOAS. Deriving this representation is straightforward and mostly involves remembering the labels of actions that were active when animating the OOAS. Also, test case generation (TCG) always is bound by an upper limit on the search depth.

The random strategy only involves a random walk over the original model. It is the cheapest way to produce test cases and does not rely on mutants. The sole complication is with non-deterministic models that yield possibly tree-shaped, non-linear, test cases containing inconclusive verdicts next to the standard pass and fail. A model is said to be non-deterministic if, given a unique current state and some input, it can move to more than one possible next state. In other words, the SUT is free to choose a next state. This means that during TCG MoMuT::UML needs to either insert inconclusive verdicts for allowed observables that branch away from the chosen random sequence when it wants to produce a linear test case, or explore all observables and generate tree-like adaptive test cases. Notice that adaptive test cases will also include inconclusive verdicts for observables that lead away from the test goal. MoMuT::UML tries to end each random test case with an observable for better testability. While being cheap, the random strategy cannot guarantee a certain test coverage.

When using the mutation strategy MoMuT::UML needs to decide whether the behaviour of the mutant differs from that of the original and generate a test case leading to this difference. To decide whether the behaviours are different, MoMuT::UML mainly employs Tretman’s ioco relation [12] relation. Another option, exclusively offered by the symbolic back end, is a refinement relation defined over the state [13]. Notice that the former implements so called strong mutation testing (the difference must be observable), while the latter corresponds to weak mutation testing (a difference in the internal state suffices) [13]. By default MoMuT::UML uses the ioco relation which is defined over traces of LTSs (SUT, Model) as follows.

\[ \text{SUT} \xrightarrow{ioco} \text{Model} = \forall \sigma \in \text{traces}(\text{Model}) : \text{out}(\text{SUT after } \sigma) \subseteq \text{out}(\text{Model after } \sigma) \]

Here after denotes the set of reachable states after a trace \( \sigma \), and out denotes the set of all observable events in a set of states. The observable events are all output events plus one additional quiescence event for indicating the absence of any output. The ioco relation supports non-deterministic models (cf. the subset relation) as well as partial models (only traces of the Model are tested) but does not lend itself easily to compositional techniques [14]. For input-complete models ioco is equivalent to trace-inclusion (language inclusion). To compute the ioco conformance, MoMuT::UML builds a product-LTS from the mutant (“SUT”) and the original (“Model”) on the fly. If an ioco violation is found, a test case is extracted from the LTS. This step is usually done with multiple worker processes in parallel, each worker processing one mutant at a time. In contrast to the random strategy, the mutation strategy is expensive but guarantees a fault coverage of the model.

Finally, MoMuT::UML offers a combined random and mutation strategy. If selected, the tool will first generate a small set of random tests and use this set to filter the mutants: any mutant that is detected by the random tests is removed from the set and only the ones not detected remain. Next, MoMuT::UML runs the mutation-based strategy on the reduced set of mutants. Past evaluations have shown this strategy to deliver test-suites with the best detection rates (cf. [2], [5]).

MoMuT::UML produces positive, abstract test cases in the Aldebaran\(^3\) format. The test cases are abstract as they are:

\(^3\)http://www.inrialpes.fr/vasy/cadp/man/aut.html
on the same level of abstraction as the UML model. Positive means that only all the expected, correct behaviour is specified in the test. Consequently, each test case can be seen as an LTS with explicit pass and inconclusive verdicts and implicit fail. Depending on the model and the capabilities of the test case generation back end, the test cases are either in linear form or adaptive, i.e. graphs. Before the test cases can be executed on the SUT, they need to be made concrete, which means that any abstraction made in the UML model needs to be undone so the test matches the interface of the SUT. Please note that this step can be fully automated.

III. UML Models For Test Case Generation

Figure 2 shows a typical UML test model. It comprises a class diagram, an instantiation diagram, and a state machine. Notice that in the class diagram the AlarmSystem class is marked as system under test, while the other classes are marked as environment. Both stereotypes are provided by the TCG profile shipped with MoMuT::UML and are used to define the testing interface: whenever the AlarmSystem calls a method of/sends a signal to an object part of the environment, this action is assumed to be observable. In return, whenever the environment sends a signal to or calls a method of the AlarmSystem object it will be deemed a controllable event.

In testing there is a principle distinction between state-based versus event-based testing methodologies. MoMuT::UML falls into the latter. It is assumed that the system under test communicates via events with the tester. This setting has certain implications. One is that, in the model, non-void method calls between the SUT and the environment have to be treated specially when generating the OOAS: they need to be split into two consecutive events. Whenever the SUT calls a method of the environment that returns a value, MoMuT::UML will generate a sequence of an observable and a controllable event. Similarly, when the environment calls a method on the SUT that returns a value, it will yield a controllable followed by an observable event. MoMuT::UML supports parametrised events, which are not used in Figure 2a, as the car alarm system model is too simplistic. That said, MoMuT::UML will automatically add one parameter to each event when generating the OOAS in case the UML model uses timed triggers. The new parameter holds the time when an event has to occur.

Currently, MoMuT::UML supports UML types that can be mapped to integers, such as integer, boolean, and enumerations. In addition it supports aggregate types like classes, signals, and lists (cf. Table I). MoMuT::UML also supports arbitrary associations and a limited version of sub-typing. In essence late binding, overriding, and overloading are not available. Due to the ambiguous semantics, inheritance of state machines is not supported. Only leaf classes in the inheritance hierarchy are allowed to own a state machine.

TABLE I: List of supported UML elements

<table>
<thead>
<tr>
<th>Types</th>
<th>Classes</th>
<th>Active/Passive Associations</th>
<th>Restr. Inheritance</th>
<th>Member Fields</th>
<th>Signal Receptions</th>
<th>Methods Def. + Body</th>
<th>OCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enum</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List/Set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Passive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>−</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>oclAsType</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>and, or, not, implies, missionary, for, in, with, out</td>
<td>exists, forall, select, at, size</td>
<td>oclInState, oclIsKindOf, oclIsTypeOf</td>
<td>oclAsType</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3: Graphical User Interface

On the other hand, MoMuT::UML also honours SysML requirements and annotations and allows mapping of tests to requirements. There are only a few elements that are currently not supported like history states. Instantiation is done via a separate Instance Diagram (cf. Figure 2b) that specifies all objects and their links. Notice that MoMuT::UML was primarily designed with embedded control systems in mind, hence the dynamic creation and destruction of objects needs to be emulated.

The preferred UML editor is Eclipse Papyrus (Kepler), although MoMuT::UML has a limited ability to work with UML models coming from Visual Paradigm 10.2. MoMuT::UML also comes with a small graphical user interface (cf. Figure 3) that allows the monitoring of the test case generation progress.

A. Model Mutation

Currently, the mutation engine directly works on the UML model. Table II lists the set of mutation operators the user can choose from. Past evaluations have shown that some of the operators may overlap, i.e. sometimes produce syntactically equivalent mutants. Therefore it is recommended to use a subset for test case generation, which we are currently developing guidelines for. We are also looking into the set of mutations offered: past evaluations have shown the current set not to be optimal, in the sense that it produces many mutants that yield the same test cases. For the upcoming MoMuT::UML version we want to improve the situation and move to a more efficient engine offering a refined set of mutation operators.
which means that only one object remains. It operates on preprocessed and fully flattened OOADs, explicitly interpreting two action systems, one marked as original, one marked as mutant and generates the corresponding LTSs. While the LTSs are generated, a synchronous product modulo \( \text{lcoo} \) is computed on-the-fly. This computation also includes a determination of the LTSs, in which silent (= hidden) transitions are removed. Silent transitions often occur in action systems generated from UML models as they are used for model-internal computations that encode the UML semantics.

While constructing the synchronous product in a breadth-first manner, \( \text{lcoo} \)-conformance is checked implicitly. Whenever non-conformance is detected, a so-called fail state is added to the product LTS. In general, the maximum exploration-depth, i.e. the number of visible actions on a path beginning from the initial state, is limited by some user-defined number. Once Ulysses has found a fail state, however, it will restrict further exploration to the depth of the fail state found. This optimisation was implemented in order to limit the test case generation time of complex systems to reasonable durations [5]. Notice that further exploration is conducted as Ulysses always compiles the full product graph. Finally, test cases are generated by choosing a trace within the product graph starting from the initial state and ending at a fail state. Action systems usually describe non-deterministic behaviour, i.e. there can be multiple observable actions enabled at the same time and an implementation is allowed to choose among them. So whenever there is a branch on observables in the product graph along the chosen trace to the fail state, this branch also is copied into the generated test case, yielding

### IV. Enumerative Back End

The first backend that is offered by MoMuT::UML is an explicit \( \text{lcoo} \) conformance checker called Ulysses (cf. [5], [16], [17]). It operates on preprocessed and fully flattened OOADs, which means that only one object remains. Ulysses simultaneously

![Fig. 2: MoMuT::UML Test Model of a Car Alarm System](image-url)

**TABLE II: List of supported UML mutation operators**

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>Upper Bound on # of Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change AGSL Int. Literal</td>
<td>( O(v) ) ( v \ldots ) literals in AGSL statements</td>
</tr>
<tr>
<td>Change OCL Int. Literal</td>
<td>( O(v) ) ( v \ldots ) literals in OCL expressions</td>
</tr>
<tr>
<td>Change Time Trigger Val.</td>
<td>( O(t) ) ( t \ldots ) transitions with time triggers</td>
</tr>
<tr>
<td>Invert Change Expr.</td>
<td>( O(t) ) ( t \ldots ) transitions with change trigger</td>
</tr>
<tr>
<td>Invert Guard</td>
<td>( O(t) ) ( t \ldots ) transitions with guard</td>
</tr>
<tr>
<td>Invert Change Expr.</td>
<td>( O(c) ) ( c \ldots ) boolean OCL sub expressions</td>
</tr>
<tr>
<td>Remove AGSL Stmt</td>
<td>( O(a) ) ( a \ldots ) AGSL statements</td>
</tr>
<tr>
<td>Remove Call Trigger</td>
<td>( O(t) ) ( t \ldots ) transitions with call trigger</td>
</tr>
<tr>
<td>Remove Change Trigger</td>
<td>( O(t) ) ( t \ldots ) transitions with change trigger</td>
</tr>
<tr>
<td>Remove Effect</td>
<td>( O(t) ) ( t \ldots ) transitions with effect</td>
</tr>
<tr>
<td>Remove Entry Action</td>
<td>( O(s) ) ( s \ldots ) states with entry actions</td>
</tr>
<tr>
<td>Remove Entry Action</td>
<td>( O(s) ) ( s \ldots ) states with exit actions</td>
</tr>
<tr>
<td>Remove Signal Trigger</td>
<td>( O(t) ) ( t \ldots ) transitions with signal trigger</td>
</tr>
<tr>
<td>Remove Time Trigger</td>
<td>( O(t) ) ( t \ldots ) transitions with time trigger</td>
</tr>
<tr>
<td>Replace Effect</td>
<td>( O(t \cdot t) ) ( t \ldots ) transitions, ( t \ldots ) trans. w. effects</td>
</tr>
<tr>
<td>Replace Entry Action</td>
<td>( O(a \cdot a) ) ( a \ldots ) entry actions, ( a \ldots ) other entry actions</td>
</tr>
<tr>
<td>Replace Exit Action</td>
<td>( O(a \cdot a) ) ( a \ldots ) exit actions, ( a \ldots ) other exit actions</td>
</tr>
<tr>
<td>Replace OCL Enum. Val.</td>
<td>( O(e) ) ( e \ldots ) enum. literals in OCL expressions</td>
</tr>
<tr>
<td>Replace OCL Operator</td>
<td>( O(o) ) ( o \ldots ) ( o \ldots ) in OCL expressions</td>
</tr>
<tr>
<td>Replace Signal Event</td>
<td>( O(t \cdot s) ) ( t \ldots ) signal triggered trans., ( s \ldots ) signals</td>
</tr>
<tr>
<td>Set Guard to False</td>
<td>( O(t) ) ( t \ldots ) transitions</td>
</tr>
<tr>
<td>Set Guard to True</td>
<td>( O(t) ) ( t \ldots ) transitions with guard</td>
</tr>
<tr>
<td>Set OCL Sub-Expr. False</td>
<td>( O(e) ) ( e \ldots ) boolean OCL sub expressions</td>
</tr>
<tr>
<td>Set OCL Sub-Expr. True</td>
<td>( O(e) ) ( e \ldots ) boolean OCL sub expressions</td>
</tr>
</tbody>
</table>
an adaptive test case. Only if no path leads to the fail state, an inconclusive verdict is added to the test case. Figure 4 shows a test case generated from the deterministic car alarm system model. Since this UML model includes time triggers, the events are parametrised with the first (and in this case only) parameter being the amount of time after which the event needs to be given (for controllables) or has to occur (for observables).

To optimise the number of test cases generated, Ulysses first checks whether an already existing test is able to detect the non-conforming behaviour of a mutated action system. This is done by computing a synchronous product between the test case and the associated LTS of a given mutant. Only if none of the test-mutant product LTSs reveals a difference in behaviour, i.e. no test kills the mutant, the full mutation-based test case generation is started.

The back end supports a rich set of data types, such as booleans, integers, lists, and tuples. These data types cannot only be used to define state variables but also for parameters of actions and functions. Due to Ulysses' support of complex data types no further restrictions to the supported UML elements are necessary. However, since Ulysses enumerates all possible traces of actions, it is quite essential to restrict the range of the integer types used as rigorously as possible. Especially if there are states with a lot of enabled actions or if there are parametrized actions, the state-space explosion problem occurs rather easily. Notice that parametrized actions are automatically included in the OOAS as soon as time triggered transitions occur in the UML state machine.

V. Symbolic Back End

In order to cope with more complex models a second back end was implemented. It tackles the challenge of parametrized actions by using a symbolic approach, based on Microsoft’s SMT solver Z3® [18]. The back end relies on the assumption that solving constraints with an optimised SMT solver is more efficient than enumerating all possible values in practice.

In addition to ioco the back end supports a conformance relation based on UTP refinement [19]. This conformance relation is based on a predicative semantics, i.e. the behaviour of an action in the OOAS is expressed as a predicate over variable valuations before and after the action has been executed [20]. According to this relation, an implementation refines its specification if the predicate encoding the implementation implies the predicate encoding the specification. So, in order to generate a distinguishing test case, the symbolic back end checks whether the mutant model refines the original model and if a non-conforming state is reachable from the initial state of the system. Both checks are encoded as satisfiability problem and solved using the SMT solver. The predicative semantics also encodes which action is executed including all its parameters.

One difference between this conformance relation and ioco is that refinement is more strict than ioco as any difference in the variable valuation causes non-refinement [21]. In ioco only labels are considered and an implementation is allowed to react to additional, unspecified, input labels and can still conform. The latter allows ioco to handle partial models. We consider using refinement as weak mutation testing, since any difference in the state variable valuation is sufficient for non-conformance and ioco as strong mutation testing, since introduced faults have to be propagated until a wrong output occurs.

The symbolic back end also allows to combine the two conformance relations, in which case refinement is checked first. This singles out equivalent and conforming mutants. Only if non-refinement is identified, an ioco check is performed. Note, that this ioco check does usually not start at the initial state, but instead at the state, in which the refinement was violated. This is a good trade-off between the better performance of the refinement check and the better fault-finding power of strong mutation testing.

The back end currently supports the data types boolean and integer. Complex data types such as lists and tuples have not been implemented yet. That said, finite lists that occur in the UML model are flattened to multiple OOAS-variables in a preprocessing step and are thus eliminated. Unfortunately this may result in huge test models and, hence, limits the applicability of the back end to UML models with one reactive instance as the system-under-test. The reason for this is that the asynchronous communication between objects is translated into list operations when transforming the UML model to an action system. Another limitation of the symbolic back end is that the generation of adaptive test cases has not been implemented and linear test cases are created instead. Whenever the system specification is non-deterministic in the sense that the implementation is allowed to choose among different output events, an inconclusive verdict is given immediately instead of trying to reach the fail state anyway.

http://z3.codeplex.com
In summary, our symbolic back end is highly optimised for test models with parametrized actions [7]. This applies to most models in which time constraints are considered. Models comprising multiple objects, however, have to be refactored before the symbolic back end can be applied.

VI. Lessons Learnt When Applying MoMuT::UML

Over time MoMuT::UML has been applied to a number of cases studies of increasing complexity. The start marks the MOGENTES car alarm system, also used in this paper, featuring one live state machine, an OOAS model of approximately 900 LoC, and an explicit state vector the size of about 0.3 kB. The next step, as supported by the current stable version, were models with a couple of live state machines, about 2000 LoC OOAS, and an explicit state vector of around 1 kB. Models this size can be found when testing measurement devices, see [2]. For the next version of MoMuT::UML we target models with about 150 parallel running state machines, leading to an OOAS the size of about 10 000 LoC and an explicit state-size of about 20 kB. Early development versions of the upcoming MoMuT::UML are already able to produce random tests and run mutation analyses on models this size. So, including the current development version, we have three very different test-case generation back ends and a number of different case studies available to draw conclusions from.

A. MBMT Methodology

The main finding after applying MoMuT::UML to several case studies is that mutation-based test case generation lives up to its promise and delivers strong test suites: they are compact, both in the number and length of the test cases, and have an excellent fault detection capability. For example, in [2] MBMT-generated tests not only found new bugs in software components several years in use but were also efficient in discovering manually seeded faults. Often the fault detection capabilities can be further improved by combining MBMT with random testing (cf. [2], [5]). While this combination might reduce the compactness of the test suite, both strategies seem to complement one another very well in general. Unsurprisingly then, our combined strategy often found the most bugs.

We also did a few tests with pure random TCG: after generating a comparatively large set of random tests we ran the mutation analysis on the full set of model-mutants. To our surprise the many random tests proved quite good at revealing model-mutants and often killed more than 70% of the mutants. Hence this is another argument for the combination of random and mutation-based test case generation strategies.

An outcome of the evaluation in [2] was that test cases for 5% of the mutants covered the faults of an additional 40%, which means that we should be able to significantly reduce the number of mutants and still maintain fault coverage on the type of models investigated. Put differently, there is a large and unexploited potential in reducing the computational cost of MoMuT::UML’s MBMT strategy.

In one particular setting we faced a test automation system that had to query the SUT for state changes and also was limited in the amount of queries it could make during some interval of time. In this situation the information about the faulty behaviour of the mutant, which MBMT provides, proves handy as it allows the test automation system to concentrate on fault-relevant queries only. However, it is clear that in doing so one reduces the fault detection capabilities towards the modelled faults and might miss others.

B. Modelling Languages

An interesting lesson we learnt was that UML’s state machine language is not as easily comprehensible as one might hope. Initially chosen due to the high interest of the industry partners in MOGENTES, UML was less intuitive to use than we had expected. In part this is due to the modelling tools lacking usability, but engineers also complained about the subtleties of the notation itself. Hence, the feedback we got from our case studies suggest that a specially tailored domain specific language (DSL) might be preferable in most cases. Also, from the implementation’s point of view, UML is not an easy target to support. Its lack of formal semantics and the general openness makes UML state machines highly modelling tool dependent. We also had to introduce such tool-dependencies ourselves: we chose our own action language, as no standard one was available at the time. Also, in order to support partial test models we had to deviate from the standard UML semantics a little bit. Unlike the UML standard, MoMuT::UML does not treat state machines as input complete specifications. In particular the UML standard enforces a kind of angelic completion of state machines as events that cannot trigger a transition will be automatically dropped. This is the same as adding a transition that self-loops with the ignored event as trigger. In MoMuT::UML the modeller has to explicitly create such self loops, otherwise the behaviour is assumed unspecified.

The second modelling language MoMuT::UML supports is object oriented action systems. OOASs are quite powerful and allow for a relatively simple mapping of UML. That said, the UML to OOAS conversion often times produces less than optimal OOAS code due to requirements stemming from the event-based state machine semantics. Due to its expressiveness, some OOAS language features easily lead to the state-space explosion problem. Two are especially noteworthy. First, OOASs are asynchronous and use non-deterministic choice of actions to emulate parallel running processes. Given a model with over 100 state machines running in parallel, the possible number of action-interleavings skyrockets. Hence, some dynamic partial order reduction is mandatory and can achieve astonishing results as we see in our development version. Lacking this algorithm, the current stable version of MoMuT::UML can handle smaller models only. Second, OOASs allow the modeller to ”let the system search for a permitted value” (projection). For example, when looking for the point in time when the next event will fire, an enumerative approach will have to enumerate all time-progression values until either it is out of range or has found a permitted value. To counteract this issue we have developed the symbolic back end that drastically improves the situation. Unfortunately, however, the current implementation cannot handle lists very well, hence UML operators over sets need to be ”flattened”, which can lead to a significant increase in the OOAS size up to the point where the back end solver cannot handle the symbolic transition system any more. We deem the way out of this dilemma to be a proper combination of enumerative and symbolic approaches, which we are currently working on.
C. The Implementation

MoMuT::UML is distributed as an executable .jar file and builds on the Eclipse platform (esp. the EMF, UML, and OCL frameworks). All parts of MoMuT::UML that are concerned with language parsing, handling of UML, OOAS generation, and job management are implemented in Java 7. The implementation heavily relies on the visitor pattern, a lot of Eclipse’s libraries, and ANTLR [22]. This turned out a good choice, although we run into performance and memory issues with some of the Eclipse modelling libraries in the mutation engine when working on large UML models. At the time of writing, the Java code responsible for the UML to OOAS conversion and the UML mutation engine comprises about 10k LoC (not counting the libraries MoMuT::UML depends on). Adding the remaining modules – including the ANTLR generated OOAS parser – brings the total amount of Java code to about 55k LoC.

Both back ends of the current stable version were written in Sisctus Prolog, with the symbolic back end interfacing to Microsoft’s Z3 solver after the Prolog integrated one could not handle the given formulae any more. While Prolog is fine for rapid prototyping it does have significant limitations. For one, Prolog does not have a nice way of multi-threading. MoMuT::UML works around this issue by running several back end-processes in parallel. For another, we found our enumerative back end’s Prolog implementation needing too much memory and not being fast enough. Hence we moved to C++ and a full just-in-time compilation via LLVM [23] for the back end we are currently working on. At the time of writing, this new back end comprises about 25k LoC C++ code.

VII. RELATED WORK

Jia and Harman [24] give a detailed survey on publications about mutation testing techniques since Lipton introduced the idea in 1971 [25]. They also state that there was comparatively little work done on using mutations to drive test case generation; this concept was initially described by Ammann, Black and Majurski [26].

Work on mutation coverage driven TCG picked up in the recent years. For example, the work of Papadakis et al. [27], [28] is concerned with automated white-box TCG for programs. The authors rely on symbolic/concolic execution, mutant schemata and the weak mutation testing criterion to lower the cost of mutation-based TCG. While we deem weak mutation testing not strong enough in our black-box and event-based setting, MoMuT::UML will most likely benefit from techniques like concolic execution and mutant schemata. The latter term refers to meta-programs that embody all mutated versions, saving mutant compilation time. Incidentally, we are incorporating a technique similar to mutant schemata in the upcoming MoMuT::UML version. One important additional difference between our setting and the one of Papadakis et al. is that we deal with highly non-deterministic models, while sequential programs usually are deterministic.

A lot of work has been done on test case generation from UML including UML state machines (e.g. for integration testing by Ali, Briand et al. [29]; achieving control and data flow coverage by Kim, Hong, Cho, Bae and Cha [30]) but mutation was usually only used to evaluate the results and not for driving the TCG. MBMT, i.e. model-mutation driven TCG, was applied to probabilistic and stochastic finite state machines by Hierons and Merayo [31] and to abstract state machines (ASM) by Gargantini [32]. However, these works seemingly were not taken further towards industrial applicability.

Fraser and Gargantini [33] evaluated various coverage criteria to drive test case generation via model checking. The mutation coverage in their comparison provided a high cross coverage with the other coverages used, while producing a low to average number of tests of low to average length. A major result of this comparison was that there was no single superior coverage criterion subsuming all others - regarding coverage, combinations of criteria showed to be the best strategy. Taking this into account, the architecture of the upcoming MoMuT::UML version has been adapted to easily add support for additional coverage criteria.

VIII. CONCLUSION AND FUTURE WORK

We have presented MoMuT::UML, a black-box test case generation tool that specialises in fault-based test case generation. More precisely, MoMuT::UML generates model mutants and finds test cases that reveal them. To our knowledge, it is the first and up to now only tool doing so for UML state machine models. The generated test cases are both effective in finding faults and efficient when executed.

We have already shown over several case studies the value of the generated tests. In this paper we shared an outlook of our planned future work, driven by the lessons learned from the case studies. In particular the upcoming MoMuT::UML 3.0 will feature an LLVM-based enumerative TCG back end, use dynamic partial order reduction algorithms, and feature a new dynamic and context sensitive mutation engine that will address most of the shortcomings identified in the current version. Early development versions have already shown their ability to handle the mutation analysis of networks of 150 UML state machines or an OOAS model of about 10k LoC. Beyond the scope of MoMuT::UML 3.0, we are looking into combining explicit and symbolic techniques to further improve the tool’s performance. In parallel we also want to add further test case generation strategies and supported coverage criteria.

Since the effort for building the test model and the achieved model quality are a crucial factor regarding industry adoption of the approach, we are working on support of the complete feature set for multiple input model formalisms, in particular support for domain specific modelling languages (DSL) - extending and maturing the MoMuT tool family. Further, we will explore options to support engineers in efficiently building the test models they need. While most of these options lie in the hands of the providers of the model editors, online animation and model analysis building on the execution engines of MoMuT might additionally improve the situation.

The current version of MoMuT::UML is available for free from its homepage at www.momut.org for academic and evaluation purposes. Future versions will be available from there too, as well as an online TCG service currently under development. We plan to open source MoMuT::UML once we have reached version 3.0 and invite the community to take part in our vision of bringing efficient and effective model-based testing to the average engineer.
REFERENCES


