Mutation-driven Test Case Generation Using Short-lived Concurrent Mutants

First Results

Willibald Krenn[1] and Rupert Schlick[1]

AIT Austrian Institute of Technology GmbH,
first.second@ait.ac.at

Abstract. In the context of black-box testing, generating test cases through model mutation is known to produce powerful test suites but usually has the drawback of being prohibitively expensive. This paper presents a new version of the tool MoMuT::UML, which implements a scalable version of mutation-driven test case generation (MDTCG). It is able to handle industrial-sized UML models comprising networks of, e.g., 2800 interacting state machines. To achieve the required scalability, the implemented algorithm exploits the concurrency in MDTCG and combines it with a search based generation strategy. For evaluation, we use seven case studies of different application domains with an increasing level of difficulty, stopping at a model of a railway station in Austria's national rail network.

1 Introduction

For reactive systems, especially in the safety critical domain and in contexts where updates are expensive, proper testing is mandatory. In the former domain, safety standards actually require certain forms of testing to be conducted and some standards highly recommend the use of formal methods and model-based testing. Mutation-driven test case generation (MDTCG), which is a form of fault-based testing [13], could help in satisfying the requirements posed by the standards – if it worked well enough with industrial sized models.

In this work, we address pushing out the boundaries of what can be done with MDTCG by addressing performance issues caused by the specific nature of the model-based mutation testing problem. To do so, we take advantage of the locality of mutations for the exploration itself and use mutation based heuristics for guiding a search based approach over mutant schemata. As a benchmark we use seven industrial sized UML models of different domains. Notice that we could not handle most of these with our previous versions of MoMuT::UML[1] [1]. The models were built either by industry engineers at partner companies or in close cooperation with them. The smallest model describes the behaviour of a typical car alarm system, while the largest is a model of a mid-sized railway station in

[1] Pronounced as "MoMuT for UML".

Table 1: Three Generations of MoMuT::UML

	Mutator			Test Case Generator			
Engine	Mutants	Level	Compression	Search	Detection	Exploration	Execution
Gen1	F	UML	-	BF	S	Enum.	Interpreter
Gen2	F	UML	-	Re	S, W	Symb.	SMT
Gen3	F, H	OOAS	Schemata	GR	S, W	EPOR	Compiled

F . . . . . First Order Mutants
H . . . . . Higher Order Mutants
OOAS . . . . . Object Oriented Action System
GR . . . . . Guided Random
BF . . . . . Breadth First Search
Re . . . Symb. Refinement Check + Reachability
S . . . . . Strong Mutation + IOCO
W . . . . . Weak Mutation
EPOR . . Enumerative Partial Order Reduction
SMT . . . . . Satisfiability Modulo Theories

the national rail network of Austria. The models have different characteristics: some favour symbolic approaches, some favour enumerative ones, some are highly concurrent, some are serialized, some use discrete time, while others do not.

Table 1 provides an overview of the three MoMuT::UML generations and their differences. The first generation appeared around 2010 and was based on a Prolog-based enumerative input-output conformance (ioco)[17] checker, called Ulysses, that was paired with a separate UML mutation engine. Generation one proved powerful enough to handle simple to slightly complex models [3]. The second MoMuT::UML generation changed the exploration approach from interpreted enumeration to SMT-solver-based exploration. Although the SMT-based back end proved to be more efficient than the first generation engine and made a big performance impact with a particular type of model, its lack of list-support and object variables meant it could not handle the models we were ultimately aiming for. That said, we applied it successfully to a use-case that is also re-presented in this paper. The article [2] provides the following summary of our findings when applying the second generation MoMuT::UML:

> The figures on the computation time prove that we spend more than 60% of the total TCG time checking equivalent model mutants. . . . The data also demonstrates that our mutation engine needs to be improved to generate more meaningful mutants as . . . 80% of the model mutants are covered by test cases generated for 2% of them.

Back then, the total TCG time was in excess of 44 hours for test cases with a depth of 25 interactions generated from an UML model comprising one instance, and 64 attributes. Our goal was to handle models in excess of 2000 instances, 3000 attributes, and to produce vastly deeper test cases in less time. So, for MoMuT::UML 3.0 we switched from formal conformance checking to a search-based exploration strategy with short-lived mutants as announced in [3]:

> . . . the authors attempt to combine directed random exploration and mutation based test-case generation on-the-fly. The idea is that during

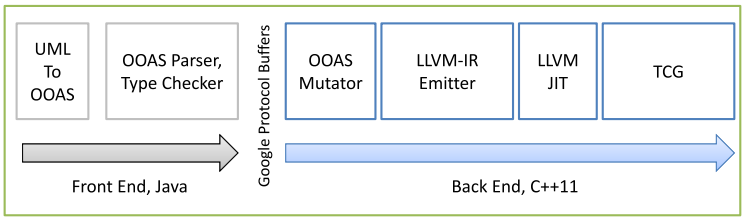


Fig. 1: MoMuT::UML 3.x Principal Architecture

> the exploration of the system, mutations which are close to the current state are dynamically inserted and conformance is checked. ...

The contributions of this paper are threefold. First, we describe the techniques used in the latest version of MoMuT::UML that allow us to scale up MDTCG to industry relevant model sizes. Second we present four case studies taken from industry the first time and, third, we evaluate our approach according to the following research questions. *RQ1:* Can we apply MDTCG to industrial-sized models at a cost (time, resources) the user would be willing to accept? *RQ2:* Do we retain enough fault coverage for the tool to be useful? *RQ3:* Given our two different guidance heuristics, are both equally suitable?

The paper is organized as follows. Section 2 introduces the mutation-based test case generation problem, mentions the main difficulties associated with it, provides a list of mutation operators used for the presented experiments, and gives a brief introduction of the Object-Oriented Action Systems modelling language. Section 3 presents MoMuT::UML's main system architecture, discusses the mutation engine in detail, and introduces the short-lived-mutant algorithm. Section 4 then presents the seven case studies and is followed by an evaluation in Section 5. The paper closes with a discussion of related work in Section 6 and concludes in Section 7.

2 Generating Tests from Model Mutants

Mutation-driven test case generation strives to automatically produce test cases that are able to detect faulty ("mutated") versions of a given specification ("model"). Research has shown that this methodology is powerful and can subsume other coverage criteria, like condition coverage [15], given the right set of mutation operators. Besides requiring a model, the main drawback of MDTCG is the computational complexity involved in finding test data that is able to reveal a faulty model ("mutant"). This challenge is easily cast as a model-checking problem, however, using model-checkers for test generation has its own issues [6], e.g., with large and non-deterministic models.

Besides the general computational complexity of the underlying problem, another issue that further increases the computational cost is that of equivalent

Table 2: OOAS Mutation Operators

Mutation Operator	Number Of Mutants, Upper Bound		
Disable Guarded Command	$O(g)$	$g\ldots$	# guarded commands
Replace Binary Operator	$O(b)$	$b\ldots$	# of $\geq, >, <, \leq, =, \neq, +, -, *, div,$ $mod, and, or, \implies, \in, \notin$
Replace Unary Operator	$O(u)$	$u\ldots$	# of $-, +, abs, not, \forall, \exists$
Invert Boolean-Literal	$O(x)$	$x\ldots$	# of true/false in rhs of assignments
Replace Integer-Literal	$O(x)$	$x\ldots$	# of literals in rhs of assignments

mutants. Equivalent mutants are mutants that do not show any difference in behaviour. In other words, they are observationally equivalent to the original model. These mutants, which cannot lead to a test case, carry the maximum penalty for TCG approaches based on exhaustive exploration. For example, in our previous MoMuT::UML version, we spent 60% of the overall computation time in checking equivalent mutants.

As a black-box TCG tool, MoMuT::UML requires a strong mutation testing setting, which means that it can only consider a mutant detected ("killed") if the mutant showed a difference in the observable behaviour compared to the original. The conformance relation we use for testing is Tretman's input-output conformance[17], which is defined over labelled transition systems and says that any observation that can be made from the system under test (SUT) after a particular trace must be predicted by the model. Notice that if the SUT only produces a subset of the possible observables, it is still fine. For a formal definition of ioco and LTSs, we refer to Tretmans [17].

Figure 1 shows an overview of MoMuT's architecture. The tool is distributed as single .jar file and combines a front end, written in Java, with a back end that is written in C++11. While the front end is responsible for checking the input, the back end is doing the actual work. Notice that front- and back end are separated via Google Protocol Buffers, so it is possible to use the back end stand-alone. As input, MoMuT::UML may use UML state charts or Object-Oriented Action Systems (OOAS). If a UML model – comprising state charts, class diagrams, and instance diagrams – is used, MoMuT first translates it into OOAS code [10]. The next step is to add mutations to the model. As can be seen in the figure, this is carried out at the OOAS level. Applying mutations directly to OOAS has the advantage of being applicable to all front-end languages (not only UML). One might see the possibility of introducing "semantic UML mutations" as a drawback but since the UML to OOAS mapping preserves the UML model's structure nicely the vast majority of mutations have a 1:1 mapping to UML. For example, the mutation operator disabling a guarded command maps to disabling transitions in the UML model. Table 2 provides an overview of the mutation operators used in the experiments described in this paper. After adding mutations, the OOAS is compiled to native machine code and finally the test case generation phase is started. Like all model-based testing tools, MoMuT delivers

```
types
    t_Status = {Ready, Busy};
    t_UserAction = {SetPause, SetStandby, StartMeasurement, StopMeasurement};
    Example = autocons system
    |[
        var
            status                    : t_Status = Ready;
            runDeviceStateChangeEvents: bool     = false
        methods
            setStatus(newStatus: t_Status) =
                requires newStatus <> status: o_statusChanged(newStatus); end // skip
            end;
            delayedActions: bool = result := runDeviceStateChangeEvents end
        actions
            obs o_statusChanged(newStatus: t_Status) = requires true: skip end;
            ctr c_userAction(action: t_UserAction) =
                requires not delayedActions():
                    requires action = t_UserAction.SetStandby:
                    /* ... */
                end;
            i_delayedDeviceStateActions =
                requires runDeviceStateChangeEvents:
                    runDeviceStateChangeEvents := false
                end
        do
            i_delayedDeviceStateActions() []
            var input: t_UserAction: c_userAction(input)
        od
    ]|
system
    Example
```

Fig. 2: Example OOAS

test cases on the level of abstraction of the model – in other words *abstract test cases*[19]. Hence before running any tests on the SUT, they might need to be concretized. As a test case output format MoMuT currently uses the Aldebaran format[2] but also writes a dot-graph in addition.

2.1 The Input Language: Object-Oriented Action Systems

OOAS are a suitable modelling language as they have formal semantics, are relatively simple, and well suited to express discrete state transition systems. An example OOAS can be found on the web at `www.momut.org`. The language is based on a generalisation of Dijkstra's guarded command language [14] and Back's action system [5] formalism. It is similar to Event-B in some sense, although less restrictive in terms of nesting of guarded commands and actions. Figure 2 shows an example model in the syntax MoMuT::UML is parsing. Each action system may declare attributes, methods, labelled actions, and one do-od block that drives the execution. As long as one action in the do-od block can be executed, the block keeps iterating. MoMuT::UML uses several extensions to the original action system, one being labelled actions. An action can be marked

[2] `http://cadp.inria.fr/man/aut.html`

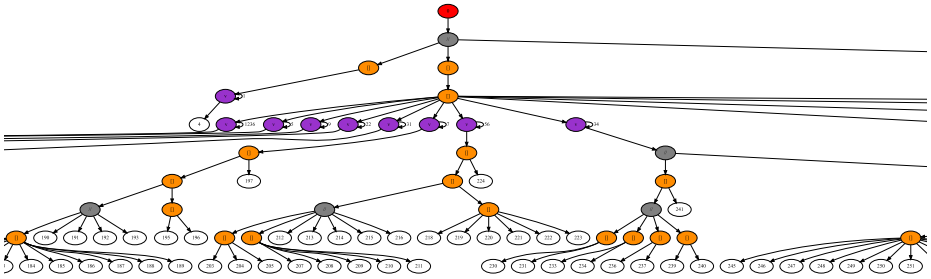


Fig. 3: Decision graph of an OOAS generated from UML

"obs" (observable), "ctr" (controllable) or neither, which means it is an internal action. Notice that we disallow recursion in OOAS and also do not provide loops (with the exception of the do-od block). Instead, the language provides a fold operator. All data types, including lists and integers, need to be declared with bounds. Direct access to attributes of other objects is forbidden. Instead, getter/setter methods are used.

Statements can be combined with sequential (";"), non-deterministic ("[]"), or prioritised ("//") composition operators. Object instantiation is only allowed upon attribute initialization as MoMuT::UML computes the set of live objects statically at compile time. Late-binding is not supported. The language features a way to express a simple scheduling policy: on a system basis, objects can be given priority ("//") over objects of other systems. A final feature of the language is called projection. Given the statement *var input: t_UserAction: c_userAction(input)* where *t_UserAction* is an enumeration with values *SetPause ... StopMeasurement*, the system will try to execute *c_userAction* with all possible values of *t_UserAction*. Put differently, it is a shorthand for saying *c_userAction(SetPause) [] ... [] c_userAction(StopMeasurement)*.

The main difference between OOAS and conventional languages is the way the sequential composition works together with guarded commands ("requires"). If we had a statement *requires (A): skip end; requires(B): skip end*, the system would first check whether A and B hold at the required times before executing the composed statement. The system, in other words, needs to compute the enabledness of actions. For model-animation this means that some sort of backtracking is mandatory. Also, as concurrency is expressed through non-deterministic choice, an efficient exploration engine is needed when computing all possible traces through the OOAS.

Given all the different ways of composing actions in non-sequential ways, it is interesting to observe how many levels deep these structures usually are nested. Figure 3 shows a cut from the decision graph taken from the big railway station model. The red node is the initial block, which is part of the scheduler. This is where execution starts. All the other, coloured, nodes represent backtracking/decision points. Grey nodes stand for prioritized composition, orange nodes

for non-deterministic composition, and purple ones for projections (the self loop indicates the amount of values to try). White nodes indicate a terminal, sequential block. As can be seen, the UML mapping does not usually produce very deep backtracking structures. However, this is a static view and does not take into account operators like fold, forall, and exists in full detail.

3 An Architecture for Short Lived Mutants

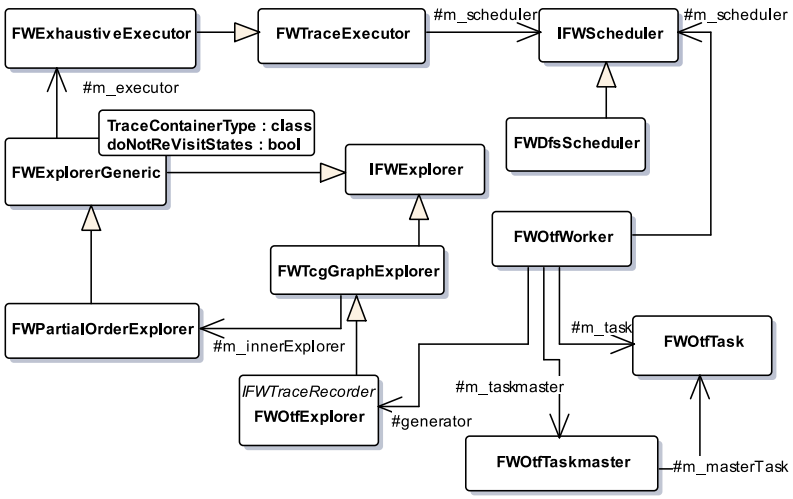


Fig. 4: MoMuT::UML 3.x TCG Architecture

Compared to earlier generations of MoMuT::UML, the key architectural changes are (a) native just-in-time compilation, (b) mutant compression through schemata (c.f. [18]), (c) adoption of partial order reduction techniques, and (d) dynamic instantiation of short-lived mutants. To facilitate these changes, the back end was completely re-written with a clear focus on performance.

A first step towards this goal is to natively execute the OOAS model. This is where MoMuT relies on the LLVM[11] compiler framework to compile the model just-in-time. Since OOAS are inherently non-deterministic and in need of backtracking, a first prototype inlined the code for the backtracking search into the LLVM-Intermediate Representation (LLVM-IR) of the model. This, however, proved quite fragile (stack saving logic, inability to compare intermediate states, fixed search algorithm), so we turned to a forward execution approach instead. Forward execution means that there is no backtracking going on inside the compiled model. Instead, during execution, the compiled model asks an external scheduler component, which is part of the runtime provided by MoMuT,

what alternative to take. This is done for one trace within the do-od blocks at a time. In case the schedule leads the execution engine to some guarded command that is disabled, the run is aborted and the scheduler informed about the result. In the simplified class diagram of Figure 4, classes *FWTraceExecutor* and *FWDfsScheduler* build this basic execution framework. Notice that the forward execution works in our case as the actions within one iteration of the do-od block of a system do not compute highly complex mathematical functions. So the time spent for re-computing partial results is negligible.

Using this simple trace-based execution framework as the basis, MoMuT::UML succeeds in adding a generic explorer (*FWExplorerGeneric*) and one that takes partial orders into account (*FWPartialOrderExplorer*) on top. To actually generate test cases, an instance of the class *FWTcgGraphExplorer* can be used. This class takes an *FWPartialOrderExplorer* and adds logic for building up a test graph. The logic that drives the short-lived mutation strategy is implemented in *FWOtfTaskmaster*. This class owns a set of workers (*FWOtfWorker*) and manages exploration tasks (*FWOtfTask*). Each worker runs in its own thread and owns an instance of an *FWOtfExplorer*, a subclass of the *FWTcgGraphExplorer*.

In total, which means including the backtracking engine, MoMuT::UML comprises about 74600 LoC Java, and 59300 LoC C++ codes. The Java-part of the OOAS compiler was recently released under BSD and is available on www.momut.org. We are currently looking into options of making the source of the C++ backend available under some academic research license.

3.1 Mutant Schemata

For our short-lived mutant project, we needed a compiled model that, at each point, included the information about possible mutants. Also, we wanted to limit our calls of the LLVM JIT compiler to a minimum and required cheap mutant instantiation at any given system state. This lead to the following design. First, the mutation engine walks over the abstract syntax tree (AST) of the OOAS before it is turned into LLVM-IR. According to selection criteria that may be set by the user, the mutation engine selects nodes in the AST and adds a mutation annotation that essentially is an alternative AST with the mutation inserted. Second, during LLVM-IR generation, a specialized code emitter looks at all the mutation annotations and emits the additional code and additional calls into the MoMuT::UML runtime library that determine whether the execution will follow the original model or any of the mutations. Technically this is done by matching mutant-IDs. The original model has the mutant-ID of 0, any real mutation a value greater than this. When instantiating the *FWTraceExecutor*, a configuration setting describes which mutation-ID should be enabled during execution. This design makes it possible to enable a set of mutations at once, effectively enabling MoMuT::UML to deal with higher order mutations. In this work, however, we always select one mutation-ID only. So in the end enabling or disabling a mutation becomes setting an integer value in the configuration of the execution engine, which facilitates rapid mutant instantiations. Also, as the compiled

Algorithm 1 TCG With Short-Lived Mutants

```
procedure GENERATE(numTests, depthTest)
    error ← 0
    test ← 0
    while test < numTests do
        master ← submitInitialTask()                    ▷ Explore original until first choice
        error ← waitForTaskCompletion()
        if error ≠ 0 then
            break
        end if
        choiceString ← ""
        choiceStates ← master.getStartState()
        choiceCount ← 0
        while choiceCount < depthTest do
            queueLiveMutants(choiceString)              ▷ Schedule running mutants
            queueNewMutants(choiceStates)               ▷ Add new mutants
            error ← waitForTaskCompletion()
            if error ≠ 0 then
                break
            end if
            filterMutants()                             ▷ Killcheck!
            selectNextInput(master, out choiceString, out choiceStates)
            choiceCount ← choiceCount + 1
        end while
        writeTest(master, test)
        test ← test + 1
    end while
    return error
end procedure
```

model asks the runtime which mutant to select, the runtime knows about all possible mutations on the current path and can decide whether to spawn a mutant or not. Notice that we require the state structure to remain unchanged between original and mutant, which restricts the set of possible mutation operators.

3.2 Test Case Generation With Short-Lived Mutants

Algorithm 1 sketches the main control loop MoMuT::UML uses to start/stop mutants and find its way through the state space. The basic pattern this algorithm follows is to explore the original model from a given state, e.g., the initial state, up to the next choice point where the tester is required to select a visible action. In other words, all internal actions and all observable actions that do not warrant a choice on part of the tester are expanded automatically. During the exploration of the original model, the number of candidate mutants is tracked.

Upon reaching the next choice point, exploration of the original model is paused and new mutants are instantiated. Mutant instantiation not only is based on the information gained when exploring the original but also takes the number of already running mutants and a user-defined limit of a maximum number of active mutants into account. After updating the list of active mutants, the system re-plays the input, the original model was faced with, on all mutants. Next, the mutants are filtered. All mutants that showed observables not predicated by the original are considered killed and removed. All mutants that crashed during execution are also removed. Also all mutants that reached identical states as the

original are removed and, finally, mutants that were live for too many steps (20) are removed and marked as "given up on". Finally, MoMuT calls on a heuristic to select a new input and explores the original model again, closing the loop.

Currently two different heuristics for selecting the next input are implemented. The first heuristics just uses a weighted random choice over the set of available actions. The heuristics guides the exploration in a sense that it penalizes already taken actions so as to prefer never-before selected actions over already taken ones. The second heuristic selects a bounded subset of available actions and explores all of them. Once done, the action leading to the most new candidate mutations is chosen. In case none is available, the heuristics regresses to the guided random choice described before.

4 Case Studies

Table 3: Properties of the Test Models

	UML		OOAS			TCG			
Model	Time	Objs	Traces	Stmnts	Attrs	State Size	Ctrs	Init-τ	Init-S
M1	Y	1	$55 \cdot 10^6$	362	34	0.3	8	$2 \cdot 10^3$	4
M2	Y	2	$108 \cdot 10^9$	748	98	0.8	8	$3 \cdot 10^3$	2
M3	Y	1	$41 \cdot 10^3$	877	64	0.5	26	$1 \cdot 10^2$	2
M4	Y	2	$3 \cdot 10^3$	1215	67	0.4	34	$3 \cdot 10^1$	2
M5	N	151	$3 \cdot 10^3$	3831	479	9.7	14	$4 \cdot 10^4$	46
M6	N	125	$2 \cdot 10^3$	4798	404	22.3	172	$1 \cdot 10^5$	82
M7	N	2847	$51 \cdot 10^3$	26385	3127	*184.9*	1652	$8 \cdot 10^7$	2572

Time ..... Whether the UML model uses timed triggers, i.e. discrete time.
Objs ..... Number of instances in the UML model; Approx. parallel running state machines.
Traces ..... Theoretical maximum for one dood block run. Fold being restricted one-element lists.
Stmnts ..... Number of OOAS-statements.
Attrs ..... Number of attributes, i.e. non-local/"class-level" variables.
State Size ..... In KiB. Memory needed to hold all attributes of all instances; M7: lower bound.
Ctrs ..... Number of controllable events in the model.
Init-τ ..... Number of traces necessary to expand all τs from the initial state, after POR.
Init-S .... Number of expansion steps necessary to expand all τs from the initial state, after POR.

Table 3 provides a comparison of some of the key properties of the test models considered in this paper. Models M1 and M3 were already described in previous publications [1,2,3] and remain unchanged, while M2 is a new model for a use case also described in [3]. The table is separated into three main columns, reading *UML*, *OOAS*, and *TCG*. The properties listed under the *UML* column concern the UML model itself: whether the model uses timed triggers, and how many objects are instantiated. For the majority of the models, the number of objects corresponds to the number of parallel running state machines, with the

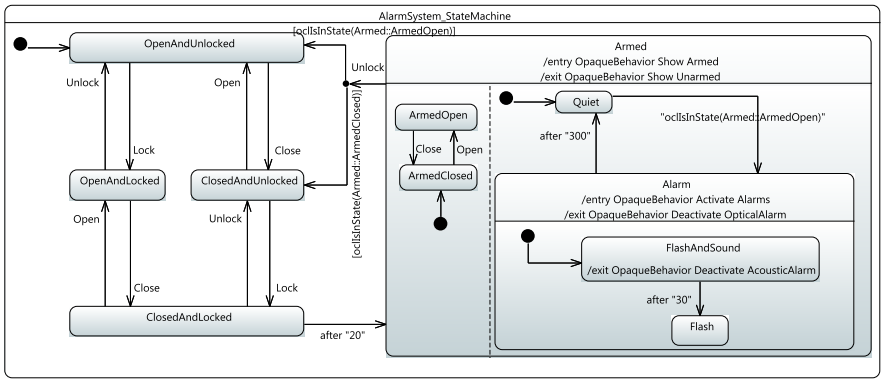


Fig. 5: Statechart of M1

exception of M5 that also instantiates non-active classes. Please note that we only use UML state charts, class diagrams, and instance diagrams in this work.

Column *OOAS* in Table 3 lists key properties of the transformed UML models. In particular it shows the number of different execution traces that are possible, considering the non-deterministically composed actions of the do-od blocks. Notice that since MoMuT::UML supports lists, the OOAS models need a way of iterating over them. As we disallow recursion and loops in OOAS, this functionality is provided through a fold operator included in the language. The number of traces reported, which in general is dependent on the number of times the fold operator is applied on a list, assumes that all lists have a length of one element. The second property shown is the number of statements, as found by the OOAS-compiler. Finally, the number of attributes at a system-level is given.

Column *TCG* describes dynamic properties of the models. First, the size of the state is given in KiB. For all models, except M7 this number is computed as including the maximum length of all lists used as attributes and, hence, represents an upper bound. For M7 this number becomes meaningless (>8 MiB, more than 9000 lists), hence we give the lower bound of the state size, assuming all lists are empty. Second, the number of controllable events is reported. Controllables are the inputs the tester can give a system under test (SUT). Some models make heavy use of parametrized controllables, hence the number only is an indication of the complexities involved. In particular M2 has parametrized controllables that lead to more than 1600 possible inputs to the SUT. Third, the number of traces the test-case generation engine needs to look at when expanding all internal actions from the initial state is reported. Finally, the last column shows the number of do-od block iterations that are necessary to expand-away all internal actions starting from the initial state. Please note that the figures of the last two columns are after partial order reduction (POR). In the remainder of this section we introduce the use cases individually.

M1 – Alarm System. M1 is a simple model of a car alarm system. Previous results with earlier generations of MoMuT and the model have been described in, e.g., [1,3]. The sole complication offered by the model is the use of timed triggers which implies fold-operations over lists. It also uses orthogonal regions as can be seen in Figure 5.

M2 – Loader Bucket Implement. M2 models the control loop (including error handling) of a bucket loader implement controller. The controller receives joystick deflection values as inputs and computes output values that will drive valves controlling the movements of the bucket. Although M2 is a rather small model, it is highly complex, as can be seen in Table 3: due to heavily parametrized actions, it requires the highest number of traces for one iteration of the do-od block. In lieu of symbolic execution in MoMuT, we limited the inputs to manually defined equivalence classes. Initial findings with previous versions of MoMuT and the use case can be found in [3]. Please notice that the model used in the current work differs from the ones used in [3]: M2 is a complete model of the system. We do not resort to partial models, as we had to previously.

M3 – Measurement Device. M3 is a model of a remote control protocol of an exhaust measurement device taken from industry. Our initial findings of test case generation for M3 in an industrial context have been published previously [2]. Here, we use the same model with our new test case generation engine. In terms of complexities posed, the model is slightly more complex than M1.

M4 – Automated External Defibrillator. M4 models the diagnostic logic of an automated external defibrillator device. The properties can again be seen in Table 3: although it features more statements than M3 it is less complex.

M5 – Safety Critical Systems Middleware. M5 is a model of a subsystem of a safety critical systems middleware that is in production. In difference to the other case studies, the model makes extensive use of UML-call-triggers and is close to the actual implementation. Although it does not use timed triggers, it uses discrete time internally: with the help of parametrized actions the tester is in control of time progression. M5 instantiates a rather large number of objects but behaves rather synchronously due to the extensive use of call triggers. The computational overhead due to the low level of abstraction is significant.

M6, M7 – Railway Interlocking Systems. M6 and M7 are instantiations of a railway interlocking system. They consist of two UML models each: one shared general model that defines all classes and data structures, and one that instantiates the objects needed for the station. While M6 represents a minimal station that allows trains to pass one another, M7 is a model of a railway station located in Lower Austria. Its layout is shown in Figure 6 and comprises 37 track sections, 56 track relays, 34 switches, 22 main signals, and 145 train routes the operator can select from. M6, in contrast, only comprises 10 track sections, 4 track relays, 2 switches, 6 main signals, and 10 train routes. Both models are

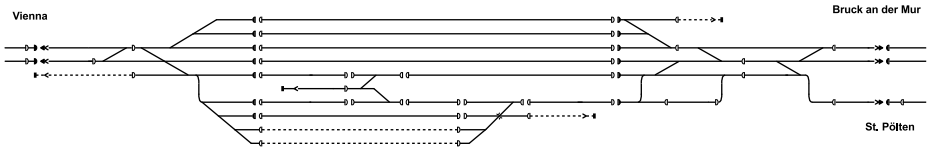


Fig. 6: Station Layout of M7

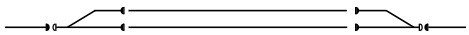


Fig. 7: Station Layout of M6

highly non-deterministic due to networks of 2847 (M7), and 125 (M6) parallel running state machines. The state machines are used to model both, physical, as well as logical entities, such as train routes. Neither M6 nor M7 includes time or parametrized actions. However, both models make extensive use of lists and forall/exists quantifiers. For example, M7 includes more than 9000 lists in the state, has more than 50 exists quantifiers and over 100 forall quantifiers that have a maximum nesting depth of five. In addition, both models have a long initialisation sequence of partially ordered observable actions before controllables appear. M7, for example, requires MoMuT to look at 83,389,266 traces only to compute the first – and most expensive – step in this long sequence. In total the initialisation sequence comprises 264 steps. Only after this initialisation sequence is complete can MoMuT start creating inputs that check the interlocking logic. Notice that M7 was directly derived from the original station data used to configure the computerized interlocking system in charge of the station. As such it represents the most complex model presented in this work.

5 Experimental Evaluation

We run the evaluation benchmarks on two different servers. The first machine is equipped with two 6-core Intel Xeon X5690 CPUs clocked at 3.47 GHz. This machine offers 24 logical cores. The second server features two 10-core Intel Xeon E-2680 v2 CPUs clocked at about 2.80 GHz and offers 40 logical cores. Both servers are backed by 192 GiB RAM. That said, we only used about 100 GiB of RAM even when running 40 workers in parallel. Due to excessive logging, MoMuT::UML proved to be I/O-bound for the small models on the small server.

Table 4 presents the main results of our evaluation. Each row stands for a TCG run that generated three test cases with a depth of 150 choices each. We chose these values mainly to be comparable to our previous publications. Unfortunately, the long running TCG process for M2 triggered a bug in MoMuT on the 40-core server which caused the tool to only generate one test case. Therefore these figures are set italic. Notice that for the larger models the generation of

Table 4: TCG Evaluation Results

			# Mutants							% Kills	
Model	Strat.	Workers	Total	Found	Exec-Err		Killed	Skipped	Given Up	Total	Found
	S1	24		303	15		272	0	16	88.3	94.4
M1	S1	40	323	303	15		267	0	21	86.7	92.7
	S2	40		303	15		261	0	27	84.7	90.6
	S1	24		1089	77		850	111	51	63.4	90.0
M2	S1	40	1417	*856*	*37*	*(5)*	*310*	*484*	*25*		
	S2	40		*996*	*27*	*(3)*	*319*	*625*	*25*		
	S1	24		1145	25		1103	8	9	86.7	98.5
M3	S1	40	1297	1145	25		1098	0	22	86.3	98.0
	S2	40		1145	25		1116	14	19	87.7	99.6
	S1	24		810	21		781	0	8	69.7	99.0
M4	S1	40	1142	810	21		775	0	14	69.1	98.2
	S2	40		835	21		801	0	13	71.5	98.4
	S1	24		635	55	(9)	389	0	191	26.8	67.1
M5	S1	40	1505	724	108	(39)	426	0	190	30.5	69.2
	S2	40		725	90	(22)	474	0	161	33.5	74.6
	S1	24		802	23		661	76	42	32.7	84.9
M6	S1	40	2044	802	23	(2)	654	41	84	32.4	84.0
	S2	40		832	23	(2)	680	40	89	33.7	84.1
	S1	24		1601	30		1496	47	28	42.8	95.2
M7	S1	40	3524	1601	37	(11)	1495	2	67	42.9	95.6
	S2	40		1751	37	(11)	1637	12	65	46.9	95.5

Rows represent the results of TCG runs that generated 3 test cases with 150 choices each. Strategy S1 represents the guided random approach, strategy S2 the not-yet-encountered mutants based one. Figures in parenthesis indicate how many of the execution errors are due to time-out. Skipped means mutants found but never scheduled. Figures in italic indicate one generated test case only.

only three test cases is expected to result in a less-than-desired total mutation coverage, as the amount of possible behaviour increases sharply. As a further remark let us say that we know to have mutations in dead code in M6 and M7. This is due to the fact that these instances build upon a shared, general railway model and our UML to OOAS translation engine is not clever enough to remove all dead code. This is also corroborated by the fact that the bigger model M7, that uses more model elements, has the better total mutation coverage than M6.

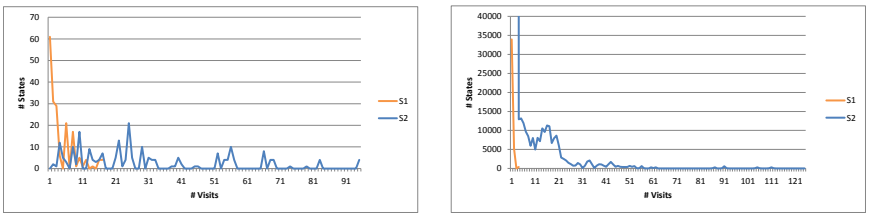
As the table shows, our proposed methodology works surprisingly well: if MoMuT is able to find a mutant, it is very likely killed. Table 5 shows a summary of how many steps are needed for a kill. The data confirms our focus on short lived mutants, as most of the kills happen early on. For the smaller models we can also report a very solid total mutation coverage often beyond 85%. These results also have to be seen in the context of the total TCG time: it took MoMuT 104 seconds to generate the test cases (S2) for M1 on the big machine with 40 workers. The tool was done with M3 in 320 seconds on the same machine, achieving a total mutation coverage of about 88%. The three tests for M7 were completed in 22.6 hours, which means it took MoMuT about 7.5 hours per test. Most of the cost actually lies in the computation of the station-initialization: as each

Table 5: TCG Time And Number of Steps To Kill

	Time (h:m:s)		S1: Killed in Step#					S2: Killed in Step#				
Model	S1	S2	0-4	5-9	10-14	15-19	20	0-4	5-9	10-14	15-19	20
M1	00:01:21	00:01:44	241	3	23	0	0	255	6	0	0	0
M2	*19:30:22*	-	*850*	*0*	*0*	*0*	*0*	-	-	-	-	-
M3	00:02:32	00:05:20	1094	4	0	0	0	1109	0	7	0	0
M4	00:02:18	00:03:16	767	0	0	8	0	788	8	0	5	0
M5	00:32:49	00:30:18	400	16	8	0	2	447	3	24	0	0
M6	00:13:24	00:22:34	647	1	4	2	0	675	2	2	0	1
M7	17:09:24	22:37:33	1493	2	0	0	0	1633	2	2	0	0

Data given for runs with 40 workers and a cut-off mutant exploration time of 2*time(orig) + 3 min, except for M2 which is data from an unconstrained run with 24 workers.

element moves into its starting position, MoMuT is faced with a huge amount of concurrency. It goes without saying that this can be easily optimized away by using the state after the initialization as initial state. Hence the figures presented here are the worst case. Compared to pure random TCG, which generates one random test with 150 choices in about 2.5 hours, strategy S2 is about three times more expensive. This is due to mutants taking longer to compute but also due to the CPU clocking lower when reaching its package power limit. Notice that we allow mutants to only take the double amount (plus three minutes) of the time the original model needed to complete the step. This is mostly to recover from mutants stuck in internal, i.e. tau, loops.



(a) M1 (b) M7

Fig. 8: Number of times states have been visited.

During the evaluation we also tracked the states MoMuT was exploring. Figure 8 shows the resulting number-of-visits vs. number-of-states graphs for M1 and M7 and both strategies. As can be seen in the figure, strategy S2 increased the number of visited states when compared to S1. In total, MoMuT explored 185 states of the original model M1 given strategy S1. This number increased to 217 states when switching to strategy S2. For M7 the figures look similar: strategy S1 made MoMuT find 39 539 unique states when exploring the non-mutated model. Switching to strategy S2 increased this number to 7 113 946 states. The latter figure seems surprising and needs additional review, however,

the experiments yielding this data were run on the big server. Hence strategy S2 was exploring 40 actions in parallel when trying to find the best next candidate action. Except for strategy S2 in M1 the plots in Figure 8 look as we would like them to be: the majority of states was only visited a couple of times.

RQ1: Can we apply MDTCG to industrial-sized models at a cost (time, resources) the user would be willing to accept? Based on the results, we think we can argue that, indeed, we are able to handle industrial-sized models at a cost that is acceptable. Even the biggest models with thousands of parallel running state machines were easily handled by one computer within a time and memory budget adequate to the model size. None of the seven use cases needed more than a day of computation time, with the majority of cases actually less than 20 minutes. Of course there always is room for further optimization, however, MoMuT::UML 3.0 proves a strong base line.

RQ2: Do we retain enough fault coverage for the tool to be useful? In case of the smaller models and the chosen test case generation settings, we retained enough fault coverage. For the bigger models, we did not achieve the amount of fault coverage we deem is necessary. Most probably, this is due to only three test cases of limited depth being generated. A more thorough analysis is needed to confirm this hypothesis and also rule out mutations in dead code skewing the result. All in all, we argue the outcome shows our approach to be effective: especially the concept of "short-lived concurrent" mutants proved its value as any mutant that was instantiated was killed with high probability. So, yes, in principle we retain enough fault coverage for the tool to be useful, albeit we need to extend the test case generation algorithm slightly so it considers the number of not-yet-seen mutations as a stopping criterion.

RQ3: *Given our two different guidance heuristics, are both equally suitable?* From the figures, it seems both heuristics behave roughly the same with S2 having only a slight advantage. This result is unexpected and needs further investigation. As S2 is quite a bit more expensive to run, it should also lead to better mutation coverage. Why this was not the case remains unclear. Hence, further research is necessary.

Threats to Validity. (A) Our results clearly depend on the selected mutation operators. For the experiments shown, we chose a standard set that should be representative. We are aware that more intricate mutants may be created. (B) The results also depend on the number of mutations in several ways (c.f.[4]): a high number of mutations increases the likelihood of finding some but it also increases the amount of mutations in dead code. (C) Our experiments are based on guided random heuristics that may perform differently depending on the random number generator. (D) Knowing the needed depth to achieve a high kill-ratio can shift the balance. We have tried to find some middle ground by using a depth that should cover the simple models nicely but is not enough for the bigger models. (E) Although we present seven quite different case studies, this

is still a sample and might not be representative (enough). (F) Software bugs. To reduce any potential issue related to bugs, we carefully checked the results, re-run outliers under close supervision and with different settings, and scanned the log files for output we could not explain and that needed investigation.

6 Related Work

The most recent work that is close to ours, is the article of Just et al. [9]. In their work, the authors optimize the mutation testing problem of Java programs. One of their contributions is a dynamic pre-pass that checks which mutant-test combination can be safely left out of actual test execution runs. To do so, the authors instrument the original Java program, run the tests and use a runtime library to track whether, e.g., an expression within the program would compute a different value given the presented test inputs and mutation operators. Based on this information, the authors can decide to not run certain test-mutant combinations. The authors also use mutant schemata to reduce the compile time of the mutants. Infected states, as Just et al. call local modified states of the mutant, are also checked for in MoMuT::UML. Once MoMuT::UML has instantiated a mutant it will check at the next choice point whether the state is different to the original or not. Based on this result MoMuT will keep the mutant or remove it from the set of live mutants.

The MuTMuT tool[7] also tries to optimize the mutation testing problem. Given a test suite and a multi-threaded program it tries to learn which mutations the tests are able to reach. Similar to MoMuT it uses mutants that are started from a given state of the original program.

As stated in the survey of McMinn et al. [12], work done in the general area of search based testing picked up in recent years. For example related to mutation testing, Papadakis et al. show in [16] that a basic hill climbing algorithm already leads to good results for mutation based test case generation of C programs. We also want to point the reader to an excellent survey of mutation testing in [8].

7 Conclusions and Outlook

We presented the third generation of MoMuT::UML that concentrates on scaling up MDTCG to industrial-sized use cases. We demonstrated the applicability with the help of seven UML models, the largest being a model of a railway station and comprising over 2800 parallel running state machines. Answering our research questions, we have shown that we can apply MDTCG at acceptable cost (time and resources) on the basis of the case studies. We also demonstrated that MoMuT::UML is able to achieve a good fault coverage provided the number and depth of the test cases match the complexity of the model. That said, we also identified the need to improve our guidance heuristics especially for complex models. From the two proposed guidance heuristics, S2 had a slight advantage over S1 on the more complex models but this was not a decisive advantage.

Compared to previous generations of MoMuT::UML, the presented version is a big leap ahead. It achieves comparable mutation coverage on the models previous generations were able to handle in less time and also succeeds in handling models hundreds of times more complex. Future tool-improvements will concentrate on including more expensive, i.e. formal, approaches to boost the total mutation detection ratio once the presented heuristics become ineffective. It is planned to integrate symbolic exploration techniques and to add further search strategies and fitness functions like distance metrics to the tool. During the evaluation, we also identified possible task-scheduling improvements in the presented algorithm.

Acknowledgments

The authors want to acknowledge the partners from industry supplying and helping to build the models. Especially to mention are Werner Schütz, Peter Tummeltshammer and colleagues. This paper would not have been possible without their continued support. The authors also want to acknowledge Bernhard Aichernig's group at Graz University of Technology for their continued cooperation. The research leading to these results has received funding from the European Union's Seventh Framework Program (FP7/2007-2013) for CRYSTAL Critical System Engineering Acceleration Joint Undertaking under grant agreement # 332830 and from the Austrian Research Promotion Agency (FFG) on behalf of the Austrian Federal Ministry for Transport, Innovation and Technology.

References

1. B. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. MoMut::UML model-based mutation testing for UML. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–8, April 2015.
2. B. K. Aichernig, J. Auer, E. Jöbstl, R. Korošec, W. Krenn, R. Schlick, and B. Schmidt. Model-based mutation testing of an industrial measurement device. In M. Seidl and N. Tillmann, editors, *Tests and Proofs*, volume 8570 of *Lecture Notes in Computer Science*, pages 1–19. Springer International Publishing, 2014.
3. B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. Killing strategies for model-based mutation testing. *Software Testing, Verification and Reliability*, Feb. 2014.
4. P. Ammann, M. E. Delamaro, and J. Offutt. Establishing theoretical minimal sets of mutants. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 21–30, Washington, DC, USA, 2014. IEEE Computer Society.
5. R.-J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *PODC*, pages 131–142. ACM, 1983.
6. G. Fraser, F. Wotawa, and P. Ammann. Issues in using model checkers for test case generation. *Journal of Systems and Software*, 82(9):1403 – 1418, 2009.

7. M. Gligoric, V. Jagannath, and D. Marinov. MuTMuT: Efficient exploration for mutation testing of multithreaded code. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 55–64. IEEE, 2010.
8. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.
9. R. Just, M. D. Ernst, and G. Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 315–326. ACM, 2014.
10. W. Krenn, R. Schlick, and B. K. Aichernig. Mapping UML to labeled transition systems for test-case generation - a translation via object-oriented action systems. In F. S. d. Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *LNCS*, pages 186–207. Springer, 2009.
11. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.
12. P. McMinn. Search-Based Software Testing: Past, Present and Future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 153–163, Mar. 2011.
13. L. Morell. Theoretical insights into fault-based testing. In *, Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, 1988*, pages 45062–, 1988.
14. G. Nelson. A generalization of Dijkstra's calculus. *ACM Trans. Program. Lang. Syst.*, 11(4):517–561, 1989.
15. A. J. Offutt and J. M. Voas. Subsumption of condition coverage techniques by mutation testing. Technical report, Dept. of Information and Software Systems Eng., George Mason Univ., 1996.
16. M. Papadakis and N. Malevris. Killing mutants effectively a search based approach. In M. Virvou and S. Matsuura, editors, *Knowledge-Based Software Engineering - Proceedings of the Tenth Conference on Knowledge-Based Software Engineering, JCKBSE 2012, Rodos, Greece, August 23-26, 2012*, volume 240 of *Frontiers in Artificial Intelligence and Applications*, pages 217–226. IOS Press, 2012.
17. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.
18. R. H. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using mutant schemata. In *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '93, pages 139–148, New York, NY, USA, 1993. ACM.
19. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.