Model-based, mutation-driven test case generation via heuristic-guided branching search

Andreas Fellner
AIT Austrian Institute of Technology,
TU Wien
Vienna, Austria
andreas.fellner@ait.ac.at

Willibald Krenn
AIT Austrian Institute of Technology
Vienna, Austria
willibald.krenn@ait.ac.at

Rupert Schlick
AIT Austrian Institute of Technology
Vienna, Austria
rupert.schlick@ait.ac.at

Thorsten Tarrach
AIT Austrian Institute of Technology
Vienna, Austria
thorsten.tarrach@ait.ac.at

Georg Weissenbacher
TU Wien
Vienna, Austria
georg.weissenbacher@tuwien.ac.at

ABSTRACT

This work introduces a heuristic-guided branching search algorithm for model-based, mutation-driven test case generation. The algorithm is designed towards the efficient and computationally tractable exploration of discrete, non-deterministic models with huge state spaces. Asynchronous parallel processing is a key feature of the algorithm. The algorithm is inspired by the successful path planning algorithm Rapidly exploring Random Trees (RRT). We adapt RRT in several aspects towards test case generation. Most notably, we introduce parametrized heuristics for start and successor state selection, as well as a mechanism to construct test cases from the data produced during search.

We implemented our algorithm in the existing test case generation framework MoMuT. We present an extensive evaluation of our heuristics and parameters based on a diverse set of demanding models obtained in an industrial context. In total we continuously utilized 128 CPU cores on three servers for two weeks to gather the experimental data presented. Using statistical methods we determine which heuristics are performing well on all models. With our new algorithm, we are now able to process models consisting of over 2300 concurrent objects. To our knowledge there is no other mutation driven test case generation tool that is able to process models of this magnitude.

This work has been conducted within the ENABLE-S3 project that has received funding from the ECSEL joint undertaking under grant agreement no 692455. This joint undertaking receives support from the European Union's Horizon 2020 Research and Innovation Programme and Austria, Denmark, Germany, Finland, Czech Republic, Italy, Spain, Portugal, Poland, Ireland, Belgium, France, Netherlands, United Kingdom, Slovakia, Norway. ENABLE-S3 is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" via FFG project number 853308 between May 2016 and April 2019. Furthermore, this work was partially supported by the Austrian National Research Network S11403-N23 (RiSE), the LogiCS doctoral program W1255-N23 of the Austrian Science Fund (FWF), by the Vienna Science and Technology Fund (WWTF) through grant VRG11-005, and by FFG project number 845582 (TRUCONF) .

MEMOCODE '17, Vienna, Austria
 978-1-4503-5093-8/17/09…$15.00
DOI: 10.1145/3127041.3127049

KEYWORDS

test case generation, model-based testing, mutation testing, search-based testing, heuristics, parallel search

ACM Reference format:
Andreas Fellner, Willibald Krenn, Rupert Schlick, Thorsten Tarrach, and Georg Weissenbacher. 2017. Model-based, mutation-driven test case generation via heuristic-guided branching search. In *Proceedings of MEMOCODE '17, Vienna, Austria, September 29-October 2, 2017,* 11 pages.
DOI: 10.1145/3127041.3127049

1 INTRODUCTION

The aim of model-based testing (MBT) is to automatically create test cases for some system under test (SUT), based on a formal model of the system. The model is usually derived from the system's requirements, hence MBT is a way of verifying the implemented system adheres to its original specification. Another use-case for MBT, for example, is model-based development, where models are refined in an iterative fashion towards a running system. In every iteration test cases can be created, ensuring the refinement was done correctly.

Model-based mutation testing is a fault-based variant of MBT, where the generated test cases are guaranteed to detect certain faulty versions of the specification. The idea here is to prove that the SUT is free of those faults. Faulty specifications are called mutants, hence the term model-based mutation testing (MBMT). MBMT, a semantically very rich test case generation technique, is often considered infeasible due to the high overhead related to computing test sequences that detect discrepancies between the original specification and mutations of it. We present a test case generation technique for MBMT that scales to large models from industry.

Our test-case generation tool MoMuT (www.momut.org) started as a research prototype [1] and has since become a tool that our industry partners use. The performance has improved tremendously over our initial prototypes. Starting from small toy models, we are now able to process industry-size models with more than 2300 concurrently running state machines. This was made possible by a new exploration algorithm that is able to leverage the power of today's highly concurrent CPUs.

The quality of a MBMT test-case generator is measured by the number of mutations it kills. A mutation is killed if there is a path from the initial state of the model to a state that is affected by the mutation, in the sense that the mutation changes the behavior of the model at that state. The mutation based testing community differentiates between two forms of killing, weak and strong [10, 21]. Weak killing seeks for a change in state, whereas strong killing seeks for a change in observable behavior. Killing a mutant can be dissected into two phases. First a path from the initial state to a state where the mutated statement can be triggered needs to be found. Secondly, killing analysis can be performed. The focus of this paper is on the former, but we also discuss techniques and results for the latter.

The original implementation used a straight-forward algorithm that explored the state space randomly beginning from the initial state. When parallelized, this approach ignores any opportunity of coordination between the parallel random explorations. For our new implementation we adopted a version of Rapidly exploring Random Trees (RRT), an algorithm that was very successful in task planning [30]. RRT works well in our setting. It basically works by exploring short paths into the state space and then starting anew, but not necessarily from the initial state; it can restart from any previously discovered state. This allows a massive parallelization because a large number of these short paths can be explored in parallel.

In this work we compare several heuristics to choose the starting state of a new exploration path and in what direction this path should explore.

In particular we make the following contributions.

- We lift and adapt the rapidly exploring random trees (RRT) algorithm from path planning, where it is highly successful and widely used, to the discrete verification world.
- We show that through parallelization we increase search performance and test case quality.
- We propose and evaluate new search heuristics for discrete state spaces, including techniques based on distances and rarity of states.
- Using a range of case studies obtained in an industrial context, we show that MBMT scales to large models. To our knowledge no MBMT tool can deal with models of this magnitude.
- We compare the techniques using a thorough statistical analysis of how many mutants they find and show that branching search performs well.
- We show that an approach combining multiple heuristics independently discovers more mutants than each single heuristic on its own.

The paper is organized as follows. In Section 2 we introduce the syntax and semantics of our modeling language. Section 3 provides an overview over the state space exploration heuristics and Section 4 describes mutation killing. Finally, in Section 5 we show our experimental data and statistically analyze the results. Section 6 provides a small conclusion and planed future work.

1.1 Related Research

In terms of test case generation technique, our work can be placed between search-based and guided random testing. Search-based testing techniques are widely studied. A comprehensive overview is provided in [34], both for white-box and for black-box testing. A popular approach within search-based testing is casting test case generation into an optimization problem [18, 26, 42, 44, 46]. Another popular technique within search-based testing is genetic programming for gradually improving test suite quality, starting from a random test suite [16, 32, 45]. Both techniques require executing/evaluating test cases multiple times, which is prohibitively expensive on our models.

Guided random testing is performed in [31] by introducing multiple techniques to enhance random testing with static and dynamic analysis information. A classic white-box, automated test case generation tool is DART [19], which combines random testing with symbolic based guidance. Furthermore, Randoop [36, 37] uses feedback from test executions to guide random automatic generation of Java unit tests. Whereas the above ideas are applied to code, we perform test case generation on models. REDIRECT [39] applies guided random techniques to Simulink/Stateflow models.

Adaptive random testing [11, 12] aims to distribute test inputs uniformly across the input data space. This is a similar idea to RRT, which tries to uniformly cover state spaces. However, the inherent difference to our work is that in contrast to simple input values, test cases for our models are sequences of actions. Therefore, although we can characterize the state space via distance metrics, this is much harder for the input space.

There are many different test coverage metrics, such as path, branch or def-use coverage. Unlike control or data-flow oriented measures, mutation coverage is a fault based measure. The adequacy of this measure is well accepted. Like search-based testing, mutation based testing is an active area of research [13, 17, 38, 40, 41].

Mutation testing was introduced in the late 70's by Demilio and Budd et.al. [10] as a technique for measuring test case adequacy. Since then, it is an active area of research [13, 17, 38, 40, 41]. A detailed survey over methods, problems, and developments of the field can be found in [23]. The strength of mutation coverage was evaluated with empirical studies [5, 24]. It is based on the competent programmer hypothesis [10] and the coupling effect [35].

Our exploration algorithm is based on rapidly exploring random trees [30]. Originally proposed for task planning, it was used in earlier work for the testing of hybrid systems [15].

Model-based testing is a broad field. In [43] a taxonomy for model-based testing is presented and [9] is a detailed book on model-based testing of reactive systems. Many types of models have been tested, such as finite state machines [25], Simulink [39], labeled transition systems [22], SysML [20]. In contrast, we perform test case generation on action system models. Due to its expressivity the language can serve for both original and intermediary models. For example, UML state machine models can automatically be translated to this language. In [4] a testing approach via symbolic refinement on action system models is presented. However, only very small models are processed in contrast to the models presented and processed in this work.

Our implementation is based on the MoMuT tool [1], which evolved from a long series of research [2, 3, 27]. We do not know of any tool but MoMuT that is able to process action systems of the size of our benchmark examples.

2 THE MODELING LANGUAGE

Our MBT approach is based on formal models given as action systems. The action system modeling formalism was initially proposed by Back [7] and is based on Dijkstra's guarded command language [14]. Action systems provide a good compromise between expressiveness and simplicity. Furthermore, it is general enough such that we can automatically map other modeling formalisms onto it, as it has been done with UML [27].

We give a brief overview of action systems, a more in-depth description of the formalism can be found in [27].

Syntax. An action system is of the form

$$|[\mathcal{V} : \mathcal{T} \leftarrow \mathcal{I}; \textbf{do } \mathcal{A} \textbf{ od}]|$$

where $\mathcal{A}$ is an action, $\mathcal{V}$ is a finite set of variables with types in $\mathcal{T}$, initialized with values $\mathcal{I}$.

Types are either **boolean**, **enumeration**$(\{v_1, v_2, \ldots, v_n\})$, **integer**$(n, m)$ or **list**(T, m). An enumeration type is a finite set of values $v_1, v_2, \ldots, v_n$, an integer type represents all integers in $[n, m]$ and a list type represents all lists of maximum length m, the elements of which are of type T (lists can contain lists). **boolean** is a specific enumeration type with the values $\{true, false\}$ and their known semantics. All types are finite.

An action is either a **guarded command** of the form $guard \triangleright action$, **skip**, **abort**, an **assignment** of a variable to an expression evaluating to an element of its type, or the **sequential-**, **nondeterministic-** or **prioritized composition** of two actions. The $guard$ of a guarded command is a predicate over $\mathcal{V}$. Each assignment and each guarded command in $\mathcal{A}$ is labeled with a unique label ℓ.

Expressions and predicates are formed using variables and the following operators. Boolean: $\{\wedge, \vee, \rightarrow, \neg\}$, Integer: $\{<, \leq, +, -, *, /, abs, mod\}$, Lists: $\{head, tail, fold, select(.), length, concatenate\}$. Furthermore, we allow $\forall$ and $\exists$ quantification, which is a short-hand notation for a finite conjunction resp. disjunction, since all our types are finite.

Objects. We study models that are written in an object oriented extension of action systems. However, all objects need to be created statically during initialization. Therefore, for the purpose of this work, we can view objects simply as a partition of variables. For an object, the *object value* is a Cartesian product of the values of its variables, together with a unique identifier of its deriving class. A more detailed discussion of object orientation in action systems can be found in [8].

States. A mapping of (a subset of) variables to values of their respective types is called a *(partial) state* s. Since all variables must have an initial value, every action system has a unique *initial state* s_0. Given a state s and variable v, we denote the value of s in v by $s(v)$. Given a predicate P and state s, we say that s satisfies P and write $s \models P$, if P evaluates to true under assignment s.

Semantics. The semantics of an action A is defined by the successor function $su(A, l, s) \rightarrow U$. su accepts an action A, a sequence of labels l, representing a path, that is initially empty, and a state s. It returns a set of tuples U, each tuple (l', s') consists of a sequence of labels l' and a state s'. The labels describe the guards and assignments that lead from s to s'. We write $succ_{\mathcal{A}}(s)$ for the set of successor states of s ($\{s' \mid \exists l'.\ (l', s') \in su(\mathcal{A}, \text{nil}, s)\}$). We write $succ_{\mathcal{A}}(s, l)$ for the set of successor states of s following label l ($\{s' \mid (l, s') \in su(\mathcal{A}, \text{nil}, s)\}$). We use $path_{\mathcal{A}}(s)$ as an abbreviation for $su(\mathcal{A}, \text{nil}, s)$. For a tuple $\pi \in path_{\mathcal{A}}(s)$ we use the notation $\pi.l$ to refer to its path- and $\pi.s$ to refer to its state component.

We call a state s_n *reachable* if there exists a finite sequence of states $s_0, s_1, \ldots, s_n$ starting from the initial state, such that $\forall i,\ s_{i+1} \in succ_{\mathcal{A}}(s_i)$. We call an action A *reachable*, if there exists a reachable state s, such that there is a $\pi \in path_{\mathcal{A}}(s)$ and $A \in \pi.l$.

Table 1: Successor State Semantics

Action (A)	Notation	su(A, l, s)
skip	**skip**	$\{(l, s)\}$
abort	**abort**	$\emptyset$
Assignment	$\ell : v \leftarrow e$	$\{(l::\ell, s[v := e])\}$
Guarded command	$\ell : g \triangleright A_1$	$(s \models g)\ ?\ su(A_1, l::\ell, s)\ :\ \emptyset$
Sequential Comp.	$A_1; A_2$	$\bigcup_{(l', s') \in su(A_1, l, s)} (su(A_2, l', s'))$
Non-det Comp.	$A_1[]A_2$	$su(A_1, l, s) \cup su(A_2, l, s)$
Prioritized Comp.	$A_1 // A_2$	$su(A_1, l, s) \neq \emptyset\ ?\ su(A_1, l, s)$ $:\ su(A_2, l, s)$

Mutation. In this work, we study the differences between *mutated action systems* and their original versions. A mutated action system is a duplicate of a given action system, with the exception of small syntactically correct variations, i.e. *mutations*. We allow expressions used in assignments and guarded commands to be mutated and assign a unique integer *mutation id* to every mutation introduced. For an action $\ell : A$ that is an assignment or a guarded command, we denote by $m(A) = m(\ell)$ the set of mutation ids applied to action A. We call a mutation *reachable*, if its action is reachable. For action A and mutation $m \in m(A)$ the mutated version of A is denoted by A^m. Table 2 lists the types of mutation operators we consider and gives examples for a selected subset of operators. The chosen mutation operators are those standard operators from the literature [23] that are applicable to our modeling formalism. A mutated action system $\mathcal{A}^m$ is derived from the original action system $\mathcal{A}$ by inserting exactly one mutation m. The general goal of mutation testing is not only to reach mutants, but also to kill them. Our approach to killing of mutants will be discussed in Section 4.

3 STATE SPACE EXPLORATION

In this section, we describe our main algorithm to perform state space exploration on action systems. Throughout the section, we assume an action system $\mathcal{A}$ with variables $\mathcal{V}$ and initial state s_0 is fixed.

The goal of the search is to find mutations. Given a mutation m that is attached to action A with label ℓ, we say that we *found* mutation m, if we found a reachable state s, such that there exists $(\_, l) \in path_{\mathcal{A}}(s)$ such that $\ell \in l$. The overall goal of mutation testing is to construct test cases that kill mutants. Finding mutants

Table 2: Mutation Operators and Examples

Mutation	Examples		
Replace Unary Operator	$\forall x : uint_4$	$\mapsto$	$\exists x : uint_4$
	$-x > y$	$\mapsto$	$abs(x) > y$
Replace Binary Operator	$x + y$	$\mapsto$	$x * y$
	$x = 5$	$\mapsto$	$x < 5$
	$a \wedge b$	$\mapsto$	$a \vee b$
Disable Guard	$x > 5 \triangleright A$	$\mapsto$	$false \triangleright A$
Invert Sub-Predicate	$x = y \wedge a$	$\mapsto$	$\neg(x = y) \wedge a$
Replace Integer Literal	$x \leftarrow 5$	$\mapsto$	$x \leftarrow 6$
in Assignments [1]	$x \leftarrow 5$	$\mapsto$	$x \leftarrow 0$
Replace Boolean Literal	$a \leftarrow true$	$\mapsto$	$a \leftarrow false$

is the first step towards achieving this goal. During search we simultaneously perform mutant kill analysis, which will be described in more detail in Section 4.

The search is organized in exploration steps. A single exploration step starts from one state and computes all successor traces and states. To this end, the underlying do-od loop of the model is executed, guards are evaluated, and effects are calculated. Across the whole test case generation approach this is by far the computationally most expensive part.

The main source of inspiration for our search procedure was the rapidly exploring random trees (RRT) [30] path planning algorithm. RRT splits the search into small sub-searches, which we call tasks and denote them by τ in the algorithm. Each task is given a random goal-state: a state that should be reached by the exploration. Furthermore, each task is started from the previously discovered state closest to the goal, w.r.t. some distance metric (see Section 3.2), and proceeds to explore states leading closer to its goal. In order not to get stuck and suffer from bad random goal choices, the time each task runs is kept short. By remembering path prefixes of visited states, we can construct paths from the initial state to every state we discover during the search, which are then transformed into test cases (see Section 4). The main property of RRT for our problem is that it is able to reach different areas of large state spaces quickly [28]. We assume that in different areas of the state space different actions are enabled and as a consequence different mutations can be found.

We found that distance based search does not always provide satisfactory results on our discrete models. Therefore, we abstracted the successor state computation and experimented with multiple different versions of it, using the same overall search procedure.

Dissecting the search into many small searches is well suited for parallel processing. In fact our algorithm is designed and implemented in an asynchronously parallel fashion. It is asynchronous in the sense that one task does not have to wait until another task is finished, but can be started as soon as resources permit it.

3.1 Abstract Search Algorithm

Algorithm 1 shows the pseudo code of our abstract search procedure. The algorithm is abstract in the sense that it needs to be instantiated by the following parameters: MAX_STEPS is the total number of exploration steps to be performed, MAX_CHOICES is the number of exploration steps performed by a single search task, CREATE_TASK is the method to create new tasks, and SELECT_SUCCESSOR is the method to select successor states. Every task τ is represented as a tuple consisting of its current state $\tau.state$, its goal states $\tau.goal$, which is used by some successor state heuristics, and its number of steps it already performed $\tau.steps$.

The algorithm executes two main blocks of instructions in a loop until the maximum number of steps has been performed.

The first block (lines 2–5) starts new tasks. We want to start a new task initially ($c = 0$) and whenever the number of steps between the current and the last time a task was created exceeds the maximal number of steps for each task divided by 4. We delay the creation of new tasks as opposed to eagerly starting as many tasks as possible, because start states of tasks are chosen among the previously discovered states. We found a length divided by 4 to be a good compromise between parallelism and delayed starting to benefit from prior exploration. When creating a new task, we heuristically choose a start and a goal state.

The second block (lines 6–16) performs the essential part of the search, computing successors and making the decision where to go next. Please ignore line 10 for the moment, we will discuss it later. We implicitly denote the successor state and path computation by referring to the sets $succ_{\mathcal{A}}$ and $path_{\mathcal{A}}$. All successor states are added to the set of visited states S. We add every mutation that is found on one of the successor paths to the set of found mutations. Finally, we increase the task specific- and global step count by one.

Data: Visited States $S \leftarrow \{s_0\}$
Data: Running tasks $T \leftarrow \emptyset$
Data: Found mutations $M \leftarrow \emptyset$
Data: Step count $c \leftarrow 0$
Data: Task creation count $l \leftarrow 0$
1 **while** $c <$ MAX_STEPS **do**
2 **if** $c - l \geq \frac{\text{MAX\_CHOICES}}{4} \vee c = 0$ **then**
3 $l \leftarrow c$
4 $\tau \leftarrow$ CREATE_TASK
5 $T \leftarrow T \cup \{\tau\}$
6 **for** $\tau \in T$ **do in parallel**
7 **if** $\tau.steps <$ MAX_CHOICES **then**
8 **for** $\pi \in path_{\mathcal{A}}(\tau.state)$ **do**
9 $M \leftarrow M \cup \bigcup_{\ell \in \pi.l} m(\ell)$
10 *CheckAndKillMutants*(τ.state, π)
11 $S \leftarrow S \cup succ_{\mathcal{A}}(\tau.state)$
12 $s \leftarrow$ SELECT_SUCCESSOR($succ_{\mathcal{A}}(\tau.state)$)
13 $\tau \leftarrow \langle s, \tau.goal, \tau.steps + 1 \rangle$
14 $c \leftarrow c + 1$
15 **else**
16 $T \leftarrow T \setminus \{\tau\}$

Algorithm 1: Abstract State Space Algorithm

3.2 Distance Metrics

Many of our heuristics use distance metrics to guide the search. The notion of distance is also a key concept of rapidly exploring random trees. RRTs are usually applied to path planning in the plane, where

[1] We replace integers by min, max, value +1, value-1 and by 0 if it is in the domain

the notion of distance is naturally given by the Euclidean distance. We needed to find a distance metric that fits our setting, which has rich types and many dimensions, due to the high number of variables. We assume that we always compute distances over values that are of equal type, e.g. it is guaranteed that we never apply a distance function to an integer and an enumerate value.

Distance on values. Given two boolean or enumeration values a, b, we define their distance $d(a, b) := 0$ if a is equal to b and $d(a, b) := 1$ otherwise. Given two bounded integer values a, b with bounds min, max, we define their distance to be $d(a, b) = \frac{|a-b|}{max-min}$. By re-scaling integer distance by their range, distances range from 0 to 1 and they become comparable to distances on boolean and enumeration values. Given two list values a and b, we compute their distance on their recursively flattened versions. On the flattened lists, we take care to compare only related elements, inserting dummy elements, having distance 1 to all other elements, when lengths do not match.

Distance on states. Assuming we have defined a distance metric $d(.,.)$ over individual values, we lift it to states by defining

$$d(s_1, s_2) := \sqrt{\sum_{v \in \mathcal{V}} d(s_1(v), s_2(v))^2}$$

For convenience, in the following sections, we use a default state $\perp$ and define $d(s, \perp) = d(\perp, s) = \infty$ for every state s.

3.3 Create Task Heuristics

We are now ready to describe the heuristics to create new tasks. We also refer to heuristics as strategies. The main job of task creation is to pick a state to start the task from. We are free to choose any state among the previously discovered states S and assume in this section, that this set is given.

We work under the assumption that in regions of the state space that have not been explored yet, we are likely to find new mutants. The difficulty lies in concretizing the concepts "regions of the state space" and "not explored". We propose heuristics that were designed to approximate these concepts and compare against heuristics that neglect our assumption.

Table 3 provides an overview of the heuristics described in more detail below. We assume a function $rand(.)$ that returns a random element of its input set.

Init. The `Init` heuristic simply always chooses the initial state as its start. It has the benefit over other heuristics, that due to completely restarting at every task, it is more likely to make different choices on early branches in the action system. Its downside is that it is likely to repeat work that has been previously performed by other tasks.

RandSt. The `RandSt` heuristic simply chooses one random state in S.

RGoal. The `RGoal` heuristic resembles classic RRT search. It first draws a random goal state and then finds the closest state in S. When drawing a random goal, for each variable, we uniformly pick a random value of its type range. For list variables, we first randomly choose a length and then recursively create random values according to their respective types.

Table 3: Heuristics for *CREATE_TASK*

Name	Description
`Init`	return $\langle s_0, \perp, 0 \rangle$
`RandSt`	return $\langle rand(S), \perp, 0 \rangle$
`RGoal`	$goal \leftarrow rand(\mathcal{V})$ return $\langle \arg\min_{s \in S}(d(s, goal)), goal, 0 \rangle$
`PGoal`($\mathcal{U}/\mathcal{R}$)	$\tau' \leftarrow$ randomly pick from $\mathcal{U}(s)/\mathcal{R}(s)$ $goal \leftarrow$ perturb $\tau'.start$ return $\langle \tau'.start, goal, 0 \rangle$
`RoRoSt(L)`	return task created according to $L[i]$ $i \leftarrow (i + 1) \mod len(L)$

PGoal. The `PGoal` heuristic turns around the selection process used by RRT by first picking a state and producing the goal from that state. The state to start from is picked, from the set of *unique states* $\mathcal{U}(S)$ or *rare value states* $\mathcal{R}(S)$. These sets are defined as follows: $\mathcal{U}(S) = \{s \in S \mid s \text{ was inserted into } S \text{ at most once}\}$, $\mathcal{R}(S) = \{s \in S \mid \exists v \in \mathcal{V} \text{ such that } |\{t \in S \mid s(v) = t(v)\}| < \frac{|S|}{10}\}$. The intuition is that these states lie in regions of the state space that have been sparsely explored. Then a goal state is created by perturbing a fraction of the variable values.

The idea of this heuristic is that random goals might too often point towards unreachable parts of the state space and we might get stuck trying to reach such states. Perturbed goals are close to states that we have already visited, therefore, they are more likely to be reachable. However, the global property of expanding the discovered state space to a random direction might be lost due to this selection strategy.

RoRoSt. The `RoRoSt` heuristic is a meta-heuristic that creates tasks according to multiple create task heuristics L in a round robin fashion. For the experiments presented in this paper, `RoRoSt` chooses among all heuristics but `PGoal`. The idea of round robin is to combine the strengths of the heuristics L. If one task gets stuck due to a bad start decision, the next one is not likely to get stuck in the same way.

3.4 Select Successor Heuristics

In the following, we describe the heuristics to select successor states during the search. Table 4 shows the different heuristics and gives a short overview.

We assume a preliminary selection function $select(.)$ which picks one element out of a set of tuples of states and evaluations. As instantiations for this function, we experimented with `greedy`, simply taking the best evaluation, `weighted`, picking probabilistically according to the distribution given by the evaluations, and `bucket` first grouping states into buckets of equal evaluation, secondly picking a bucket probabilistically according to the distribution given by the evaluations and finally picking a state randomly within the bucket.

RandNe. The `Rand` heuristic simply chooses randomly among all possible successor states.

Table 4: Heuristics for *SELECT_SUCCESSOR*

Name	Description
`RandNe`	return $rand(succ_{\mathcal{A}}(s))$
`Dist`	$\Lambda = \{(t,\lambda) \mid t \in succ_{\mathcal{A}}(s), \lambda = d(t, \tau.goal)\}$ return $select(\Lambda)$
`Part`	$\Lambda = \{(t,\lambda) \mid t \in succ_{\mathcal{A}}(s), \lambda = \alpha(t)\}$ return $select(\Lambda)$
`BFS`	Every 10'th choice: schedule all $s \in succ_{\mathcal{A}}(s)$ Otherwise: schedule according to `Dist`
`RoRoNe(L)`	return state selected according to $L[i]$ $i \leftarrow (i+1) \mod len(L)$

Dist. The `Dist` heuristic assigns the distance to goal state as state evaluation and picks a successor state via the *select*(.) function described above. This heuristic resembles classic RRT search

Part. The `Part` heuristic assigns the *new object states* function $\alpha(.)$ as state evaluation and picks a successor state via the *select*(.) function described above. $\alpha(.)$ returns the number of object values, c.f. Section 2, that occur in s, but not in S.

The idea of this heuristic is to have a fine grained characterization of new information in states. For large models, the search algorithm constantly finds new states, since the number of combination of variable valuations is high. It is reasonable to assume that objects of the same type typically have a symmetric role, in the sense that it does not matter which particular instance is in a state that might trigger new behavior. The `Part` heuristic exactly tries to capture and quantify the novelty of information a state supplies.

BFS. The `BFS` heuristic works a bit different than the other heuristics. It selects successors according to the `Dist` heuristic, except that in every 10th iteration all successor states are explored. One successor is picked to be explored by the current task, and in all other states new tasks are started. We expect that `BFS` will explore several (very different) directions towards a goal. We only perform the complete successor search every 10th iteration in order not to flood our search with too many tasks and intermediate states.

RoRoNe. The `RoRoNe` heuristic is a meta heuristic, assigning to every task a strategy in a round robin fashion from a list of other select successor heuristics. The strategy is fixed for one task for its whole lifetime. Similarly to `RoRoSt` for task creation, we want to benefit from the advantages of all other heuristics and avoid to get stuck during the search.

4 MUTATION KILLING & TEST CASE GENERATION

Although it is not the main focus of this work, we perform kill analysis in addition to reachability analysis. Ultimately, the goal of mutation testing is to create test cases that guard against errors, modeled by (killed) mutations. In this section, we describe how we achieve this goal by creating test cases and perform mutation killing during state space exploration.

Mutation Killing. In the realm of mutation analysis for test case generation, there exist two ways to kill a mutant. A mutant is weakly killed [21], if it produces a different state than its original version. In our setting, this means that for an action A, mutation $m \in m(A)$ and state s we have that $su(\mathcal{A}^m, \text{nil}, s) \not\subseteq su(\mathcal{A}, \text{nil}, s)$. A mutant is strongly killed [10], if it produces different output than its original version. Actions can be annotated with a special label that marks them as being observable, which corresponds to program output. We have implemented a prototypical mode that performs strong kill analysis with the help of these observable actions. However, due to the high costs associated with strong killing, we consider only the weak killing model in this work. This is a common approach to mutation testing [23].

Mutation Killing and Test Case Generation During Search. Function *CheckAndKillMutants*, depicted in Algorithm 2, performs both mutation killing and test case generation. The function is called in line 10 of Algorithm 1 for every successor transition $s \xrightarrow{\pi.l} \pi.s$ discovered during the search. Algorithm 2 records every edge in graph E that is later used to find a path from the initial state to the mutant (line 2).

Then we check for every mutation m whether the set of successors in the mutated system following labels $\pi.l$ is a subset of the successors in the original system following labels $\pi.l$. We also consider the mutant killed if it can not execute labels $\pi.l$ in state s. If this is not the case, we found a killing edge and we generate a test case by finding the shortest path in E to s and adding the last transition to $\pi.s$ (line 6). In K we collect the mutations already killed to prevent double killing, which would lead to additional test cases with arguable value.

Finally all test cases are collected in Σ. The resulting test cases are sequences of labels $\langle l_1, \ldots, l_n \rangle$ such that $s_0 \xrightarrow{l_1} s_1 \cdots \xrightarrow{l_n} s_n$ in the original system and $s_0 \xrightarrow{l_1} s_1 \cdots \xrightarrow{l_n} s'_n$ in the mutated system with $s_n \neq s'_n$. Test cases can be removed from Σ if they are a prefix of another test case, since the subsuming test is strictly more powerful than its prefix in terms of mutation killing.

Data: Killed mutations $K \leftarrow \emptyset$
Data: Test Case Graph Edges $E \leftarrow \emptyset$
Data: Test Cases $\Sigma \leftarrow \emptyset$
1 **Function** *CheckAndKillMutants*(s, π)
2 $E \leftarrow E \cup \{s \xrightarrow{\pi.l} \pi.s\}$
3 **for** $m \in \bigcup_{A \in \pi.l} m(A) \backslash K$ **do**
4 **if** $succ_{\mathcal{A}^m}(s, \pi.l) \not\subseteq succ_{\mathcal{A}}(s, \pi.l) \vee succ_{\mathcal{A}^m}(s, \pi.l) = \emptyset$ **then**
5 $K \leftarrow K \cup \{m\}$
6 $\Sigma \leftarrow \Sigma \cup \{\text{shortestPath}(E, s_0, s) \xrightarrow{\pi.l} \pi.s\}$

Algorithm 2: The *CheckAndKillMutants* function used in Alg. 1

5 EXPERIMENTS

In this section, we present the benchmark models and the experimental evaluation of the presented methods on these models. Furthermore, we discuss implementation caveats and engineering techniques we use to be able to process the largest of our models.

Table 5: Properties of the Test Models

Model	Traces	Obj	GC	Vars	SS	LoC
AlarmSystem	$55{\cdot}10^6$	3	362	42	0.3	967
Debounce	$52{\cdot}10^0$	3	35	29	0.1	655
Defibrillator	$3{\cdot}10^3$	4	1215	67	0.4	2949
Measurement	$41{\cdot}10^3$	3	877	55	0.5	1834
Loader	$108{\cdot}10^9$	4	748	98	0.8	2009
MMS	$2{\cdot}10^3$	125	4798	1490	22.3	8281
LBT	$51{\cdot}10^3$	2373	26385	27959	184.9	33289

5.1 Models

Table 5 provides a comparison of some of the key properties of the test models considered in this paper. The models were collected from several industrial use cases. The use cases for AlarmSystem, Measurement and Loader are described in [1–3]. All models were originally written in UML and automatically translated to action systems. In this work we consider only the resulting action systems. Therefore all properties of the models described here relate to the action system representation.

The **Traces** column shows the number of different execution traces that are possible, considering the non-deterministically composed actions of the do-od blocks. Notice that this is a theoretical value as not all traces may be feasible due to guarded commands not being enabled. The **Obj** column shows the number of objects in the action system. Each object has its own independent do-od loop and can act concurrently to all other objects. Although not every object is always allowed to make a transition in every iteration, this property gives an indicator the amount of concurrent actors in the respective model. The **GC** column shows the number of guarded commands in the do-od block. The **Vars** column shows the total number of variables of the action system. The **SS** column shows the size of a single state given in kilobytes, i.e. the maximally required memory to represent one state of the action system. This value is calculated assuming the maximum length of lists and therefore represents the upper bound of the state size (except for LBT where we chose to report the lower bound as the maximum would have been too much of an overestimation). Unfortunately, we can not report reachable state space size. We tested the reachable state space for the medium sized MMS and found 2.5 million states before running out of memory. Finally, the **LoC** column shows the number of lines of code in the ooas model.

AlarmSystem. AlarmSystem is a simple model of a car alarm system.

Debounce – Signal Debouncing Algorithm. Debounce models a debouncing algorithm used in the domain of safety critical industrial control.

Defibrillator – Automated External Defibrillator. Defibrillator models the diagnostic logic of an automated external defibrillator device.

Measurement – Measurement Device. Measurement is a model of a remote control protocol of an exhaust measurement device.

Loader – Loader Bucket Implement. Loader models the control loop (including user feedback and 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.

MMS, LBT – Railway Interlocking Systems. MMS and LBT are instantiations of a railway interlocking system. The original UML models consist of two parts each: one shared general model that defines all classes and data structures, and one that instantiates the objects needed for the station. While MMS represents a minimal station that allows trains to pass one another, LBT is a model of a mid sized, real life railway station. Both models are highly non-deterministic due to networks of 2373 (LBT), and 125 (MMS) concurrently running objects. The objects are used to model both, physical, as well as logical entities, such as train routes. Both models make extensive use of lists and forall/exists quantifiers. For example, LBT 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 these models have an initialization phase where all elements concurrently move into their initial position and communicate it to the outside world. In total the initialization sequence for LBT consists of 264 steps. This means that starting test case generation from the initial state is a bad choice for these models, which is confirmed by our data. The difference in number of mutants is explained by the different station layouts using different parts of the modeled functionality.

All these models have been built by researchers in cooperation with industry partners for their real use cases. For example Debounce is used in production by an industrial control systems provider to generate test cases.

The selected models provide different challenges for our test case generator. In terms of complexity, LBT is the biggest example because of its high number of concurrent objects, with MMS following with some distance. Debounce is the simplest, but still uses an integer to model discrete time. The latter is used in all models except the railway interlocking ones. Loader's challenging complexity stems from both its use of many concurrent timers and the large allowed value ranges for its input parameters, increasing the state space.

5.2 Implementation

The algorithm and heuristics presented in this work are implemented in the back-end of the MoMuT tool. In order to have a hard grip on resource consumption, the back-end of the tool is written in C++. Furthermore, the original- and mutated action systems are just-in-time compiled [6, 29] to machine code, which allows us to execute transitions fast and leverage compiler optimizations.

As mentioned in earlier sections, we implemented the search procedure in an asynchronous parallel way. There is a central scheduler, which accumulates and distributes data and performs the SELECT_START computations. Additionally, there is a set of workers that perform the labor intensive job of SELECT_SUCCESSOR computations and kill-checking. Workers store gathered data in buffers, that are repeatedly harvested by the central scheduler. Workers request new tasks from the scheduler actively. Therefore, our parallel computation scheme has two synchronization points between scheduler and workers: the harvesting of buffers and obtaining new tasks. In between these synchronization points, all threads run asynchronously.

5.3 Experimental Setup

In the following, we present the experimental evaluation of the presented techniques on our benchmark models, compare our heuristics to each other, and evaluate which ones work best, overall and for specific scenarios.

We performed our experiments on 3 servers running Debian Jessie. They have between 24 and 64 logical cores and between 48 and 192GB of RAM. In total we continuously utilized 128 cores for over two weeks to gather the data that went into this paper. We conducted 1963 experiments using different combinations of parameters and heuristics. The scripts to reproduce these experiments are available on request.

5.4 Choices of Parameter Values

We ran the algorithm with a number of different values for the following parameters and present our summary here.

MAX_STEPS. We evaluated different values for MAX_STEPS and found that runs performed with values below 150 are too fragile for random decisions and produced high variance in the results. For values above 2000 we found that there is a saturation and no improvements were made. Therefore, we settled on values 150, 500, and 2000 for the experiments presented in the following section.

MAX_CHOICES. We found that values smaller than 50 again produced results with high variance. Furthermore, for the `Init` strategy, such small values do not make sense.

We evaluate the parameter value 50 across all MAX_STEPS values, and higher values 150 and 250 for runs with higher MAX_STEPS values.

select(.) In preliminary experiments, bucket selection was clearly superior to greedy and weighted selection, which were therefore dropped in the final experiment set.

Random Seed. Although not formally specified in our algorithm, one parameter of our procedure is the random seed used for random number generation. We ran every combination of strategies with 5 different random seeds and found that there is no significant difference between the results.

5.5 Statistical Evaluation Method

We use a one-sided Mann-Whitney U test [33] in order to evaluate performance of heuristics w.r.t. mutations found. The one-sided Mann-Whitney U test takes two sets of measurements A, B. The null hypothesis is that if we have two samples a and b drawn from A and B respectively, the outcomes $a > b$ and $a < b$ have the same probability. The alternative hypothesis is that $a > b$ has a higher probability than $a < b$. We used the Mann-Whitney U test as opposed to the stronger t-test, because it seemed that our measurements are not normally distributed, which is an assumption of the t-test. Further assumptions for these tests are independence of samples, which is true, because runs are performed independently, and measurements on an ordinal scale, which is true, because our measurement is an integer number.

We use this statistical test to determine which heuristics significantly (we use the standard $p = 0.05$) outperform others. To this end, for every model, we iteratively filter our experimental data set twice, once for the CREATE_TASK and once for the SELECT_SUCCESSOR heuristics. Starting with the whole set of experiments E, in every round of this iterative process, for each remaining heuristic C, we split the remaining set of experiments into two halves. E_C are the experiments that were performed using C and $E_{\neg C}$ are the experiments that were performed using any other heuristic. We use the Mann-Whitney U test to compare E_C to $E_{\neg C}$, using number of found mutations as our indicator. We filter out the heuristic that performed most significantly worse than its counter-set, if any, by taking $E := E_{\neg C}$ and repeat the process until no heuristic can be shown to significantly under-perform.

5.6 Statistical Evaluation Results

In Table 6 we report the results of the statistical evaluation method described above. Columns **Out** show for how many models the respective heuristic was thrown out during data-set filtering, i.e. was shown to significantly under-perform. Lower values are better here. Columns **Best** shows for how many models the respective heuristic was the last remaining heuristic in the filtered data set, when the filtering is performed beyond significance level. Higher values are better here.

Table 6: Parameter Performance Across Models

CREATE_TASK			SELECT_SUCCESSOR		
	Out	**Best**		**Out**	**Best**
`RGoal`	0	2	`RandNe`	1	3
`RandSt`	1	3	`Dist`	1	2
`PGoal`($\mathcal{R}$)	1	1	`BFS`	1	2
`PGoal`($\mathcal{U}$)	1	0	`RoRoNe`	3	0
`Init`	2	1	`Part`	5	0
`RoRoSt`	2	0			

`RGoal` and `RandSt` are the most reasonable choices in the CREATE_TASK category with `PGoal` performing reasonably. On the other hand `Init` and `RoRoSt` fall a bit short. This indicates that branching search, i.e. starting tasks from intermediate states rather than the initial one indeed performs well overall. The fact that `RGoal` could not beat random `RandSt` more decisively indicates that the random goals are not of high quality. Two possible explanations are that the generated goals are often in unreachable parts of the state space, or that the search space has a non uniform topology, i.e. even though the values permit expansion into multiple directions, the enabled actions often do not allow to lead the search towards these directions.

Within the SELECT_SUCCESSOR category, `RandNe`, `Dist` and `BFS` perform best, although every heuristic was shown to be significantly under-performing on at least one model. The most significant result is that `Part` clearly under-performs and that the intuition of capturing new information of states via the `Part` heuristic was not fruitful. Unfortunately, `Dist` was not able to beat random successor choice. Two possible explanations are again the low quality of random goals, pointing in directions that are not reachable, or that the distance metric used was not able to accurately capture similarity of states in our search spaces.

Unique Finds. Even though many strategies find a similar number of mutants, the total number discovered by all strategies combined is higher than the number of mutants found by a single strategy. This is even true if we combine the mutants found over several runs of the strategy with different random seeds. We calculated the number of mutants uniquely found per SELECT_SUCCESSOR strategy, i.e. no other strategy found them. For this evaluation, we excluded the heuristics RoRoNe and BFS that are based on other strategies. We present these numbers in Table 7. For missing models none of the strategies found unique mutants.

Table 7: Unique Kills Per Model

Model	RandNe	Dist	Part
Defibrillator	126	43	0
Measurement	0	1	0
Loader	159	25	0
MMS	0	54	0
LBT	0	23	0

Similar to the results of the statistical evaluation RandNe and Dist perform well for unique finds. The total number of unique mutants found for RandNe (285) is higher than the total number of mutants found for Dist (146). However, Dist is able to find unique mutants across more models. In three models it is the only strategy to find unique mutants. Therefore, the use of different strategies is justified as they find more mutants combined than any single strategy. It appears round robin is not sufficient to combine several strategies, it is more beneficial to conduct an entire run with a single strategy and have multiple runs. One reason for RoRoNe not utilizing this combination idea effectively is that different tasks in the same search are not independent of each other. If the search is driven towards a region of the state space that does not yield many new mutants, different search strategies might not be able to correct and steer the search in better areas. In contrast, if the searches were performed independently, one search might discover a bad region, which does not affect any of the other searches.

5.7 Branching versus Non-branching

The results presented above indicate that the branching search is effective. We wanted to evaluate this aspect more carefully by comparing our algorithm operated running in a branching versus a non-branching fashion. Non-branching search means that the whole search is performed in one single task, whereas in branching search many tasks are created and terminated during search.

In Table 8 we report the time per step for branching and non-branching search. This quotient is calculated by dividing the wall-clock time the implementation ran by the total number of steps performed in the experiment. We chose to present this quotient, to make experiments with different parameters for MAX_STEPS comparable. The results show a significant increase in speed introduced by the shift to branching search.

Apart from performance we also see an improved quality of the test cases. A test case fixes the choices for non-deterministic compositions, such that one or more states are reached that weakly kill one or more mutants. An important measure of test case quality is length. The length of a test case is measured in steps where every step corresponds to one iteration of the do-od loop. The shorter a test case the better, as shorter test cases run faster, can be executed in parallel, and are easier to debug when they fail. In Table 9 we report average test case length for branching and non-branching runs. The reason for branching search runs producing significantly shorter test-cases is that they restart from intermediate positions repeatedly. Non-branching search produces one long chain of actions exploring the state space and all test share a prefix of that chain.

Table 8: Time Per Search Step Non-Branching vs Branching

Model	Non-Branching (ms/step)	Branching (ms/step)	Decrease[2] (%)
AlarmSystem	41.4	4	90.4
Debounce	40.0	5.2	87.0
Defibrillator	71.3	10.0	86
Measurement	55.2	9.9	82.1
Loader	10999.4	1660.1	84.9
MMS	465.3	63.9	86.3
LBT	18648.0	6551.5	64.9

Table 9: Avg Test Case Length Non-Branching vs Branching

Model	Non-Branching (# test steps)	Branching (# test steps)	Decrease[2] (%)
AlarmSystem	9.9	8.4	15.2
Debounce	10	7.8	22.5
Defibrillator	47.5	19	60.1
Measurement	81	26	67.9
Loader	47.2	23.2	50.8
MMS	265.2	57.5	78.3
LBT	220.5	152.3	30.9

5.8 Mutations Found & Killed

The goal of MBMT test case generation is to find and kill mutants. In the previous sections we presented the performance of our heuristics for finding mutants. Although different heuristics find different mutants, heuristics have no impact on the killing of mutants: we follow the same approach across all search strategies. Across search strategies, the ratio of killed per found mutants did not vary. Therefore, in Table 10 we present the killing ratios for all models. Furthermore, the table shows the total number of mutants seeded per model and the average number of mutants found across all heuristics.

5.9 Threats to validity

We took great care to ensure that the conclusions we draw are based on solid data collection and statistical evaluation.

[2]Decrease is calculated as $\frac{\text{Non-Branching}-\text{Branching}}{\text{Non-Branching}}$

Table 10: Mutants and Verdicts per Model

Model	Total	Found (% to total)	Killed (% to found)
AlarmSystem	389	388.4 (100%)	198 (51%)
Debounce	569	531.3 (93%)	166.6 (31%)
Defibrillator	1653	749.9 (45%)	447.3 (60%)
Measurement	1653	1144.4 (76%)	395.7 (35%)
Loader	1772	1181.6 (67%)	597.4 (51%)
MMS	1826	1195.6 (65%)	874.5 (73%)
LBT	1897	1153.3 (61%)	625.4 (54%)

Our implementation has two sources of randomness: Random choices by the heuristics and randomness in scheduling of threads. The latter can cause different results, because some heuristics depend on the states already explored. To ensure our results are comparable we used fixed random seeds. For each combination we tried at least 5 random seeds. However, we do not evaluate the individual combinations, but aggregate them to evaluate a single aspect (only successor heuristics or only create task heuristics). This does not just include the different combinations of heuristics, but also the combinations of parameters. This yields sufficient data to draw conclusions.

To compare the mutation finding performance of heuristics we go beyond simple averaging of results. The performance is often very similar at first glance. We therefore employ statistics to test whether different heuristics are indeed performing significantly different.

Another aspect is that we did not run all combinations of heuristics. However, this does not pose a problem to our evaluation because in practice one would not combine these heuristics. For example it makes little sense to start from a random state and then evaluate the successor states by distance.

6 CONCLUSION

In this work we present an algorithmic framework for heuristic-guided branching search and an extensive evaluation of heuristics in the context of action systems. While we found that a few heuristics do not perform well in practice, the remainder are similar in number of mutants found. We demonstrated that branching, parallel search is superior to a sequential search in terms of performance and test case quality. An interesting result is that no single heuristic is able to find all the mutants that are found by the others, even when repeatedly running with different parameters and random seeds. This means that the best result is achieved by running multiple heuristics independently and combining their results.

In future work, we would like to improve random state creation using static and dynamic analysis. These techniques will allow us to approximate the reachable state space and make more reasonable goal choices. Furthermore, we want to improve the distance metric by attaching it to constraints that represent reachability conditions of mutations.

REFERENCES

[1] B. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. 2015. MoMuT::UML Model-Based Mutation Testing for UML. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on (ICST)*. 1–8. DOI: http://dx.doi.org/10.1109/ICST.2015.7102627

[2] Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. 2014. Model-Based Mutation Testing of an Industrial Measurement Device *(TAP)*. 1–19. DOI: http://dx.doi.org/10.1007/978-3-319-09099-3_1

[3] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. 2015. Killing strategies for model-based mutation testing. *Softw. Test., Verif. Reliab.* 25, 8 (2015), 716–748. DOI: http://dx.doi.org/10.1002/stvr.1522

[4] Bernhard K. Aichernig, Elisabeth Jöbstl, and Stefan Tiran. 2014. Model-based mutation testing via symbolic refinement checking. (2014). DOI: http://dx.doi.org/10.1016/j.scico.2014.05.004

[5] James H. Andrews, Lionel C. Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA (ICSE)*, Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh (Eds.). ACM, 402–411. DOI: http://dx.doi.org/10.1145/1062455.1062530

[6] John Aycock. 2003. A brief history of just-in-time. *ACM Computing Surveys (CSUR)* 35, 2 (2003), 97–113.

[7] Ralph-Johan Back and Reino Kurki-Suonio. 1983. Decentralization of Process Nets with Centralized Control. In *Proceedings of the 2nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 1983) (PODC)*. ACM, 131–142.

[8] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. 1998. An Approach to Object-Orientation in Action Systems. In *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings (MPC)*, Johan Jeuring (Ed.), Vol. 1422. Springer, 68–95. http://link.springer.de/link/service/series/0558/bibs/1422/14220068.htm

[9] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. 2005. *Model-based testing of reactive systems: advanced lectures*. Vol. 3472. Springer.

[10] Timothy A Budd, Richard J Lipton, Richard A DeMillo, and Frederick G Sayward. 1979. *Mutation Analysis.* Technical Report. DTIC Document.

[11] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. 2010. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software* 83, 1 (2010), 60–66.

[12] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. 2008. ARTOO: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*. ACM, 71–80.

[13] Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. 2007. A survey on model-based testing approaches: a systematic review. *Workshop on Empirical assessment of software engineering languages and technologies* (2007), 31–36.

[14] Edsger W Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.

[15] Tommaso Dreossi, Thao Dang, Alexandre Donzé, James Kapinski, Xiaoqing Jin, and Jyotirmoy V Deshmukh. 2015. Efficient guiding strategies for testing of temporal properties of hybrid systems. In *NASA Formal Methods Symposium*. Springer, 127–142.

[16] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In *Proceedings of the 11th International Conference on Quality Software, QSIC 2011, Madrid, Spain, July 13-14, 2011. (QSIC)*, Manuel Núñez, Robert M. Hierons, and Mercedes G. Merayo (Eds.). IEEE Computer Society, 31–40. DOI: http://dx.doi.org/10.1109/QSIC.2011.19

[17] Gordon Fraser and Andreas Zeller. 2012. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 38, 2 (2012), 278–292.

[18] Matthew J. Gallagher and V Lakshmi Narasimhan. 1997. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering* 23, 8 (1997), 473–484.

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM Sigplan Notices*, Vol. 40. ACM, 213–223.

[20] Christoph Hilken and Jan Peleska. 2015. Model-Based Testing Against Complex SysML Models. In *Formal Modeling and Verification of Cyber-Physical Systems, 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*, Rolf Drechsler and Ulrich Kühne (Eds.). Springer, 284–286. DOI: http://dx.doi.org/10.1007/978-3-658-09994-7_14

[21] William E. Howden. 1982. Weak Mutation Testing and Completeness of Test Sets. *IEEE Trans. Software Eng.* 8, 4 (1982), 371–379. DOI: http://dx.doi.org/10.1109/TSE.1982.235571

[22] Claude Jard and Thierry Jéron. 2005. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer* 7, 4 (2005), 297–315.

[23] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on* 37, 5 (2011), 649–678. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5487526

[24] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014 (FSE)*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 654–665. DOI: http://dx.doi.org/10.1145/2635868.2635929

[25] HE Koenig, Y Tokad, HK Kesavan, and others. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* 4, 3 (1978).

[26] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering* 16, 8 (1990), 870–879.

[27] Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. 2009. Mapping UML to Labeled Transition Systems for Test-case Generation: A Translation via Object-oriented Action Systems. In *Proceedings of the 8th International Conference on Formal Methods for Components and Objects (FMCO)*. Springer-Verlag, Berlin, Heidelberg, 186–207. http://dl.acm.org/citation.cfm?id=1939101.1939119

[28] James J Kuffner and Steven M LaValle. 2000. RRT-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on (ICRA)*, Vol. 2. IEEE, 995–1001.

[29] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. DOI: http://dx.doi.org/10.1109/CGO.2004.1281665

[30] Steven M LaValle. 1998. Rapidly-exploring random trees: A new tool for path planning. (1998).

[31] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-Analysis-Guided Random Testing (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 212–223. DOI: http://dx.doi.org/10.1109/ASE.2015.49

[32] Jan Malburg and Gordon Fraser. 2011. Combining search-based and constraint-based testing. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011 (ASE)*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 436–439. DOI: http://dx.doi.org/10.1109/ASE.2011.6100092

[33] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[34] Phil McMinn. 2004. Search-based software test data generation: A survey. *Software Testing Verification and Reliability* 14, 2 (2004), 105–156.

[35] A. Jefferson Offutt. 1992. Investigations of the Software Testing Coupling Effect. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (1992), 5–20. DOI: http://dx.doi.org/10.1145/125489.125473

[36] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (OOPSLA Companion 2007)*. ACM, 815–816.

[37] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering (ICSE)*. IEEE Computer Society, 75–84.

[38] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, Bernd Sostawa, Rüdiger Zölch, and Thomas Stauner. 2005. One evaluation of model-based testing and its automation. In *Proceedings of the 27th international conference on Software engineering (ICSE)*. ACM, 392–401.

[39] Manoranjan Satpathy, Anand Yeolekar, and S Ramesh. 2008. Randomized directed testing (REDIRECT) for Simulink/Stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*. ACM, 217–226.

[40] Muhammad Shafique and Yvan Labiche. 2010. A systematic review of model based testing tool support. *Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04* (2010), 01–21.

[41] Muhammad Shafique and Yvan Labiche. 2015. A systematic review of state-based test tools. *Software Tools for Technology Transfer* 17, 1 (2015), 59–76.

[42] Nigel Tracey, John Clark, Keith Mander, and John McDermid. 1998. An automated framework for structural test-data generation. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on (ASE)*. IEEE, 285–288.

[43] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability* 22, 5 (2012), 297–312.

[44] Mattia Vivanti, Andre Mis, Alessandra Gorla, and Gordon Fraser. 2013. Search-based data-flow test generation. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE) (ISSRE)*. IEEE, 370–379.

[45] Joachim Wegener, Kerstin Buhr, and Hartmut Pohlheim. 2002. Automatic Test Data Generation For Structural Testing Of Embedded Software Systems By Evolutionary Testing.. In *GECCO (GECCO)*, Vol. 2. 1233–1240.

[46] Yuan Zhan and John A Clark. 2005. Search-based mutation testing for simulink models. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation (GECCO)*. ACM, 1061–1068.