Understanding The Impact of Solver Choice in Model-Based Test Generation

Ying Meng
University of South Carolina
Columbia, SC, USA
ymeng@email.sc.edu

Gregory Gay
Chalmers and the University of Gothenburg
Gothenburg, Sweden
greg@greggay.com

ABSTRACT

Background: In model-based test generation, SMT solvers explore the state-space of the model in search of violations of specified properties. If the solver finds that a predicate can be violated, it produces a partial test specification demonstrating the violation.
Aims: The choice of solvers is important, as each may produce differing counterexamples. We aim to understand how solver choice impacts the effectiveness of generated test suites at finding faults.
Method: We have performed experiments examining the impact of solver choice across multiple dimensions, examining the ability to attain goal satisfaction and fault detection when satisfaction is achieved—varying the source of test goals, data types of model input, and test oracle.
Results: The results of our experiment show that solvers vary in their ability to produce counterexamples, and—for models where all solvers achieve goal satisfaction—in the resulting fault detection of the generated test suites. The choice of solver has an impact on the resulting test suite, regardless of the oracle, model structure, or source of testing goals.
Conclusions: The results of this study identify factors that impact fault-detection effectiveness, and advice that could improve future approaches to model-based test generation.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **System modeling languages**.

KEYWORDS

Model-Based Test Generation, Model-Driven Development, Satisfiability Modulo Theories

1 INTRODUCTION

Model-driven development is a practice where models of system behavior are analyzed in order to improve the quality of the final system. Such models [36], often expressed as state-transition systems, represent the system specifications by prescribing the behavior (the state) to be exhibited in response to input. Models allow the analysis of system requirements and behavior, the debate of design decisions, and the construction of test cases before mistakes lead to expensive or life-threatening failures. Model-driven development has been adopted in safety-critical domains such as avionic [24], automotive [25], and medical systems [39].

Models allow testing activities to begin before the actual implementation is constructed, and models are suited to the application of automated test generation techniques [24, 36]. Tests generated from these models can be used either to test the model itself or to test the real system, once constructed.

A common form of model-based test generation is based on the use of bounded model checking [20, 43], where properties expected of the model are embedded as temporal logic formulae—Boolean predicates describing states encountered during execution paths [30]. Model checking relies on the use of an *SMT solver*. Solvers explore the state-space of the model—in search of states that violate the specified properties—by casting property violation as a Boolean satisfiability problem [30]. If the model checker finds that a predicate can be violated, it produces a *counterexample*—a partial test specification demonstrating the violation.

Often, rather than being used to search directly for violations of properties, we instead embed properties that we want to show *can* be met by the model. We then negate those properties, claiming that they *cannot* be met. The model checker then attempts to produce a counterexample illustrating how the property can be met. This counterexample offers test input that can be applied to put the model into the desired state. Often, this form of test generation is used to achieve structural coverage over the model [21, 24]. By repeating this process for each goal of the coverage criterion, we can thoroughly explore the state-space of the model.

The choice of solvers is important, as each follows their own algorithm, and each may produce differing counterexamples. Each may be better suited to particular classes of problems, or types of models. In many situations, multiple solvers could be used to guide the test generation process. If only one is capable of achieving goal satisfaction, the choice of solver is clear—use the one that can solve your problem. However, with advances in both computing power and algorithm design, we increasingly encounter situations where multiple solvers can produce test cases for a given model. Consider four common solvers: Z3 [14], MathSAT5 [10], CVC4 [7], and Yices2 [15]. Each is able to handle similar model structures and data types. In this situation, *does it matter which solver is used?*

Even in this situation, there may still be merit in choosing one solver over another. Multiple input choices often lead to the same state. The strategy used to sample the input space differs from one solver to another, and the resulting test suites—while demonstrating the same goals—may offer different paths or choices of input in

ESEM '20, 2020, Bari, Italy

ACM ISBN AAA-B-CCCC-DDDD-E/FF/GG…$15.00
https://doi.org/10.1145/AAA.BBB

service of achieving that goal. This can, in turn, impact the fault-detection potential of the test suite [24]. In practice, the choice of solver *could* have an impact on the ability of the generated test suite to detect failures in the model or the system. To help developers choose the right solver for their task, we must understand the differences in the test cases produced by those solvers.

To examine the fault-detection capabilities of suites produced by Z3, Yices2, CVC4, and MathSAT5, we have performed test generation over 37 models. Tests were generated targeting four different coverage criteria—Branch Coverage, Decision Coverage, Condition Coverage, and Modified Condition/Decision Coverage (MC/DC). We seeded up to 500 faults—mutations—in each model, and assess the ability of the generated test suites to detect them using two different test oracle strategies—oracles based on the output variables of the model and maximal oracles that assess the values of all model variables [23]. This experiment allows us to examine the impact of solver choice on multiple dimensions—the ability of a solver to achieve goal satisfaction, and the impact on fault-detection effectiveness of the solver algorithm, types of goals, data types of the model variables, and the type of test oracle.

The results of our experiments demonstrate that:

- CVC4 and MathSAT5 fail to produce test cases for a subset of the goals on seven models (18.92%), while Z3 and Yices2 achieve satisfaction over the full set.
- The solvers yield test suites that differ in fault-detection capabilities for 18 models (48.65%). Performance differences are seen for 13 of these models across all coverage criteria.
- When using a standard output-based oracle, suites generated by CVC4 and MathSAT5 outperform suites generated by Z3 with significance. When using a maximal oracle, CVC4 and MathSAT5 also outperform Yices2 with significance.
- Solvers and generation frameworks could improve fault detection by factoring effect propagation of variables, like those associated with testing goals, into input selection.
- Complex goals constrain the solver's ability to choose input, narrowing the performance gap. Simple testing goals can be satisfied by a wider range of input, magnifying the difference in search strategies employed by solvers.
- Solvers could improve fault-detection by applying a wider variety of values, rather than favoring fixed values.
- Solvers, by design, omit unused input variables from the counterexample. The strategy used by the test generation framework to fill those gaps impacts fault detection.
- Solvers often offer parameters that can be used to control the search. These parameters could be automatically tuned by test generation frameworks.

We observe, from this study, that the choice of solver has a major impact on the resulting test suite, regardless of the oracle, model structure, or source of testing goals. The results of this study identify factors that impact fault-detection effectiveness, and offer advice that could be used to improve future solvers and future approaches to model-based test generation as well as other test generation methods that make use of solvers, like symbolic execution [49].

2 BACKGROUND

Bounded model checking is an automated approach to property verification where the state-space of a model is explored for a state that violates user-specified properties [26, 30]. Bounded model checking is based on logical propositions, often expressed in a temporal language where a formula is not statically true or false in a model. Instead, across the states of the model reached during execution, a formula may be true in some states and false in others [30]. The problem of identifying property-violating states can be expressed as a Satisfiability Modulo Theories (SMT) instance, a generalization of a boolean satisfiability instance in which atomic boolean variables are replaced by predicates expressed in classical first-order logic [8]. An SMT problem can be thought of as a constraint satisfaction problem—we seek values for the input variables of the model that lead to a state violating the stated properties. Bounded model checking is a specific form of model checking where, given a transition-based model M, a set of logical formulae f, and a user-supplied transition limit k, we can decide whether or not f is satisfiable over all transition paths of length k. If the formulae can be violated, the model checker produces a counter-example showing a set of transitions that cause the violation.

Within the bounded model checking framework, the *solver* attempts to falsify the user-specified properties by using a search strategy to explore the state space of the model. A common search strategy is the branch-and-bound algorithm [4]. This algorithm is conceptually simple: set a literal in the property to a particular value and see if the value satisfies all of the clauses that it appears in. If so, assign a value to the next variable. However, if setting a value unsatisfies a clause, then a backtracking step (a bound) is initiated and another value is applied. This process prunes branches of the formed Boolean decision tree. Another common approach, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, is similar [13]. DPLL begins by selecting a variable and applying a value to it. If this value satisfies the clause, then all clauses containing that variable are removed from the formula. If the variable is made false due to negation, the algorithm instead remove that variable from only the clause that it is negated in. This process is repeated recursively until a solution is found. This induces a domino effect—as more variables are removed from clauses, more clauses turn into unit clauses [41].

In this study, we have made use of the JKind [19, 26] model checker and four common solvers:

Cooperating Validity Checker (CVC4) is based on the DPLL(T) algorithm, an extension of DPLL designed to offer solutions for not just Boolean variables, but also for rational and integer linear aritmetic, tuples, records, inductive data types, bit-vectors, strings, and uninterupted functions [7].

MathSAT5 is also based on the DPLL(T) algorithm, and offers its own theories for arrays and floating point calculations [10]. It offers sophisticated preprocessing of Boolean formulae, and support for incremental solving.

Yices2 is based on the conflict-driven clause learning (CDCL) algorithm [48], and is able to address formulae containing uninterpreted functions, real and integer arithmetic, bit-vectors, scalar types, and tuples [15]. It also supports both linear and non-linear arithmetic.

Z3 is also based on the DPLL(T) algorithm, and supports linear and non-linear real and integer arithmetic, bit-vectors, uninterpreted functions, extensional arrays, and quantifiers [14].

While many of these solvers are based on similar algorithms, i.e., DPLL(T), each may search the state-space differently, and each offers their own theories for solving non-trivial formulae.

In counterexample-based test generation, we produce test cases intended to put the system into a set of desired states. This could be done to show that the final system conforms to the model's behavior, or to attain coverage over certain elements of the model structure. In this form of test generation, we embed a set of one or more properties into the model. These properties describe behavior we want the model to demonstrate, or a state we want the model to be in. We then negate these properties—we assert that the properties *can not* be satisfied, and ask the search algorithm to find a set of values for the input variables that *do* satisfy those properties.

For example, we have a model containing the expression: $out = (x \ and \ s)$. We then might formulate a testing goal stating that we want variable x to be false, the expression *out* to evaluate to false, and x to influence the outcome of *out* without being masked by the value of s. This leads to the property $((not\ x)\ and\ s)$. We then negate that to be $(not\ ((not\ x)\ ands))$. When a solver is able to falsify a property, it returns a partial test specification outlining how that violation can be replicated. For one or more input variables, this specification outlines a value for each specified variable. For example, the specification $m : false, false$ outlines a two-step test case (a path with two transitions) where the value of false is provided for input variable m in both steps. By providing that input, we violate the negated property, demonstrating that the original property can be met.

The test specification provided in the counterexample may not specify values for all input variables. By design, the counterexample only provides values for variables needed to reach the desired state. The other input variables are unspecified, as they will not impact the execution path. However, executing a test case still requires providing *some* value for those "don't care" variables. Generally, default values (i.e., 0 for numeric variables or `false` for Boolean variables) are used.

3 METHODOLOGY

We are interested in understanding the effect that solver choice has on the test suites produced through bounded model checking. We are especially interested in the impact of solver choice when multiple solvers are able to produce tests for the full set of satisfiable testing goals. In this situation, each solver will produce an identically-sized test suite—with one test per goal. However, as each solver brings their own theories and applies their own search strategy, the input used to address each goal may differ. Does this difference matter? Do these variations have a practical, observable impact on the ability of test suites to detect faults?

This question motivates our study. In particular, we wish to address the following research questions:

(1) Does the choice of solver influence the number of test goals that will be satisfied for a model?
(2) Does the choice of solver influence the resulting fault detection capabilities of the produced test suite?
(3) Does the source of test goals influence the capabilities of the chosen solver?
(4) Does the choice of oracle strategy influence the capabilities of the chosen solver?
(5) Do the data types of the model variables influence the capabilities of the chosen solver?
(6) Are there additional factors that influence the capabilities of the chosen solver?

To address these questions, we have:

(1) **Gathered case examples:** We have assembled a set of 37 software models written in the Lustre language (Section 3.1). Some solvers may be better at addressing numeric constraints than other solvers. Therefore, we study models with varying mixtures of Boolean and numeric data types.
(2) **Generated mutants:** We generated up to 500 mutations (faulty versions) of each model, each containing a single fault (Section 3.2).
(3) **Generated test suites:** Different sources of testing goals will impose different constraints on the produced test suites. We focus in this study on testing based on structural coverage criteria. Simple criteria, like Branch Coverage, often have fewer goals and impose fewer constraints on input selection than complex criteria like MC/DC. Some criteria may expose differences between solvers more clearly than others. Using each solver, we generated test suites intended to satisfy Branch, Condition, Decision, and MC/DC Coverage (Section 3.3).
(4) **Computed effectiveness:** We collect data on the number of testing goals satisfied and compute the fault finding effectiveness of each test suite. A test oracle is required to judge the correctness of system behavior. The oracle specifies expected values for a set of variables. We vary the size of that set, as one solver may be better at propagating faulty state to the monitored variables than others. Therefore, we assess fault detection using both a standard oracle based on the output variables and an oracle considering all program variables—a *maximally powerful* oracle (Section 3.4).

A replication package containing all models, mutants, test suites, execution traces, and fault-detection results is available from http://doi.org/10.5281/zenodo.3484641.

3.1 Case Examples

Increasingly, our society is powered by *reactive systems*—-embedded systems that interact with physical processes. Reactive systems operate in cycles—receiving new input from their environment, to which they react by issuing output. In each step, input is received, internal computations are performed sequentially, and output is produced. Within a step, no iteration or recursion is done—each internal variable is defined, and the value for it computed, exactly once. The system itself operates as an large loop.

Such systems are commonly designed using modeling languages, which are translated into C code that can be directly flashed to hardware. Models can be developed using visual notations, such as

Table 1: Benchmark example information. "i" = int, "b" = bool. Systems in italics show difference in number of solvable test goals between solvers, systems in bold show different fault-detection performance between solvers.

Model	# Inputs	# Internal Variables	# Outputs
6counter	1 (b)	4 (b)	1 (b)
CarAll	2 (b)	8 (3 i, 5 b)	1 (b)
cd	1 (i)	6 (2 i, 4 b)	1 (b)
DockingApproach	13 (9 i, 4 b)	1410 (1211 i, 199 b)	11 (b)
DragonAll	13 (1 i, 12b)	22 (7 i, 15 b)	1 (b)
DragonAll2	13 (1 i, 12b)	27 (11 i, 16 b)	1 (b)
durationThm1	5 (2 i, 3 b)	7 (5 i, 2 b)	1 (b)
ex3	2 (b)	5 (2 i, 3 b)	1 (b)
ex8	2 (b)	5 (2 i, 3 b)	1 (b)
fast_1	14 (1 i, 13 b)	19 (b)	1 (b)
fast_2	14 (1 i, 13 b)	30 (b)	1 (b)
FireFly	9 (1 i, 8 b)	17 (7 i, 10b)	1 (b)
Gas	2 (b)	8 (5 i, 3 b)	1 (b)
HysteresisAll	2 (b)	5 (2 i, 3 b)	1 (b)
IllinoisAll	10 (1 i, 9b)	16 (5 i, 11b)	1 (b)
Infusion_Manager	20 (14 i, 6 b)	861 (797 i, 64 b)	5 (4 i, 1 b)
MesiAll	4 (b)	10 (4 i, 6 b)	1 (b)
Metros1	3 (b)	16 (7 i, 9 b)	1 (b)
MoesiAll	5 (1 i, 4 b)	12 (5 i, 7 b)	1 (b)
PetersonAll	12 (b)	28 (13 i, 15 b)	1 (b)
ProducerConsumerAll	4 (1 i, 3 b)	12 (6 i, 6 b)	1 (b)
ProductionCell	3 (b)	15 (b)	1 (b)
Readwrit	9 (b)	24 (12 i, 12 b)	1 (b)
RtpAll	12 (b)	24 (9 i, 15 b)	1 (b)
Speed2	2 (b)	5 (3 i, 2 b)	1 (b)
Stalmark	1 (b)	3 (b)	1 (b)
SteamBoilerNoArr2	19 (16 i, 3 b)	3 (2 i, 1 b)	1 (b)
Swimmingpool1	8 (2 i, 6 b)	21 (13 i, 8 b)	1 (b)
Switch	3 (b)	2 (b)	1 (b)
Switch2	3 (b)	2 (b)	1 (b)
SynapseAll	4 (1 i, 3 b)	10 (5 i, 5 b)	1 (b)
Ticket3iAll	13 (4 i, 9 b)	20 (8 i, 12 b)	1 (b)
Traffic	1 (i)	3 (2 i, 1 b)	1 (b)
Tramway	4 (b)	23 (b)	1 (b)
TwistedCounters	1 (b)	4 (1 i, 3 b)	1 (b)
Two Counters	1 (b)	3 (1 i, 2 b)	1 (b)
UMS	5 (b)	39 (b)	1 (b)

Simulink[1], Stateflow[2] and SCADE [17]. They can also be directly expressed using *dataflow languages*. The models used in this experiment were developed in the Lustre dataflow language [27], a declarative programming language for manipulating streams of variable values. Lustre offers an intermediate representation between behavioral model and traditional source code that is useful for specification, design, and analysis purposes [26]. Lustre programs can be automatically generated from visual notations such as Simulink, and can be automatically compiled to target languages such as C/C++, VHDL, as well as to input models for verification tools such as model checkers. This is a syntactic transformation, and if applied to C, the results of this study would be identical.

In this study, we make use of 37 models from the open-source Lustre Benchmarks dataset[3]. This dataset has been used in previous test generation experiments [38], and includes complex models such as *Docking_Approach*, a NASA-created example that describes the behavior of a space shuttle as it docks with the International Space Station [24]. Another model, *Infusion_Manager* represents the prescription management of an infusion pump device [22–24]. Information related to each system is provided in Table 1, where we list the number and data types of of input, internal, and output variables. The models in italics differ in terms of the number of solvable testing goals for any of the criteria, and the models in bold yielded differing fault-detection results.

[1]http://www.mathworks.com/products/simulink
[2]http://www.mathworks.com/stateflow
[3]Available from https://github.com/Greg4cr/Reworked-Benchmarks/tree/SingleNode.

3.2 Mutant Generation

The following mutation operators were used in this study: arithmetic operator change (+, -, /, *, mod, exp), relational operator change (=, ≠, <, >, ≤, ≥), Boolean operator change (∨, ∧, XOR), Boolean negation, use the stored value of the variable from the previous computational cycle rather than the newly computed value, alter a constant, and variable substitution. The mutation operators used in this study are discussed at length in [42]. These mutations are similar to those used by Andrews et al., where the authors found that generated mutants are a reasonable substitute for actual failures in testing experiments [5]. Additionally, Just et al. have found a significant correlation between mutant detection and real fault detection [33].

The mutation method used is designed such that all mutants produced are both syntactically and semantically valid. That is, the mutants will compile, and no mutant will trivially "crash" the system under test. In order to control experiment costs, we do not use all possible mutants for each model. Instead, we employ the following rule-of-thumb—if a model has fewer than 500 possible mutations, we use all possible mutations. If over 500 mutations are possible, we choose 500 of them for use in the experiment. In order to select mutants, we first gather a list of all possible mutations. Then, we use the proportions of each mutation type in the full set to select the number of mutants for the reduced set of 500. Mutants of each type are then chosen randomly until the determined number are chosen for that type. This process prevents biasing towards particular types of mutations. Instead, the proportion of each fault type is maintained, despite not using the full set.

3.3 Test Input Generation

Structural coverage criteria serve as a means to determine that the structure of system-under-test—the various elements making up the model—have been thoroughly exercised by test cases. Many structural coverage criteria, defined with respect to specific syntactic elements of a program, have been proposed and studied [24]. These criteria are used both to measure suite adequacy—as a means to assess the quality of existing test suites—and as goals for automated test generation. In this study, we are primarily concerned with reactive systems. Such systems often have sophisticated logical structures. Therefore, we focus on coverage criteria defined over Boolean expressions:

Decision Coverage: A decision is a Boolean expression. Decisions are composed of one or more conditions—atomic Boolean subexpressions—connected by operators (`and`, `or`, `xor`, `not`). Decision Coverage requires that all decisions in the system under test evaluate to both the `true` and `false`.

Branch Coverage: A branch is a type of decision that can cause program execution to diverge down a particular control flow path, such as that in `if` or `case` statements. Branch Coverage is defined in

the same manner as Decision Coverage, but is restricted to branches, rather than all decision statements.

Condition Coverage: Condition Coverage requires that each condition evaluate to `true` and `false`. Note that satisfying Condition Coverage does not always imply that Decision Coverage is fulfilled as well, as tests can vary condition values without the entire expression changing value.

Modified Condition/Decision Coverage (MC/DC): MC/DC requires that each decision evaluate to all outcomes, each condition take on all outcomes, and that each condition be shown to independently impact the outcome of the decision. Independent effect is defined in terms of *masking*—a masked condition has no effect on the value of the decision; for example, given a decision of the form `x and y`, the truth value of `x` is irrelevant if `y` is false, so we state that `x` is masked out. A condition that is not masked out has *independent effect* for the decision. Showing independent impact requires a pair of test cases where all other conditions hold fixed values and our condition of interest flips values. If changing the value of the condition of interest changes the value of the decision as a whole, then the independent impact has been shown.

The MC/DC criterion is used as an exit criterion when testing software for critical software in the avionics domain, and is required for safety certification in that domain [44]. Several variations of MC/DC exist. We use Masking MC/DC, as it is a common criterion within the avionics community [9].

We use a counterexample-based test generation framework for Lustre programs used in past research on test generation[4] [21, 24, 38]. This framework generates counterexamples using the JKind bounded model checker [19, 26], then fills in default values for any variables not specified by the specification. The solver is invoked for each goal, meaning that this form of test generation will produce one test per testing goal. There may be overlap between tests in terms of goals covered, and test suite reduction is often used to narrow the size of the suite [38]. As suite reduction adds stochastic noise to experiment results, we do not employ it in this study, and instead examine the full suite.

3.4 Data Collection

In order to compute effectiveness of the generated test suites, we produce *traces* of execution by executing each test case against the original program and each mutant—recording the value of all variables at each step.

We use *expected value oracles* as test oracles [23], where the tester defines concrete, anticipated values for one or more variables in the program. We employ two oracle formulations: an *output-only oracle strategy* defines expected values for all outputs, and a *maximum oracle strategy* that defines expected values for all variables. The output-only strategy represents the oracle most likely to be used in practice, while the maximum strategy allows examination of the best possible fault-detection scenario for a generated suite [23].

To produce an oracle, we use the values of the monitored variables from the traces gathered by executing test cases on the original program, and we compare those values to those recorded for each mutant. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected. For all studied systems, we assess the fault-finding effectiveness of each test suite and oracle combination by calculating the ratio of mutants detected to total number.

[4] Available from https://github.com/MENG2010/lustre.

Table 2: Models where a solver was unable to achieve satisfaction of all solvable goals for all criteria. In each case, the percentage of satisfaction achieved is listed. Z3 and Yice2 attain full satisfaction for all models.

Model	Solver	Branch	Decision	Condition	MC/DC
DockingApproach	CVC4	33.97%	25.29%	33.97%	15.08%
	MathSAT5				15.17%
Gas	CVC4/MathSAT5		95.23%		75.76%
HysteresisAll	CVC4/MathSAT5			41.02%	48.49%
Metros1	CVC4/MathSAT5			94.19%	
PetersonAll	CVC4/MathSAT5			97.12%	97.58%
Readwrit	CVC4/MathSAT5			97.01%	95.44%
RtpAll	CVC4			94.37%	94.95%
	MathSAT5				94.50%

4 RESULTS & DISCUSSION

We divide our analysis of the experimental results as follows: (a) the ability of solvers to satisfy testing goals, (b) the impact of solver choice on fault-detection effectiveness, and (c) the factors that influence fault detection.

4.1 Ability to Satisfy Testing Goals

Our first research question differentiates the solvers in terms of their basic ability to generate test cases. Are there models where certain solvers *cannot* satisfy all solvable testing goals? For each model, solver, and coverage criterion, we have measured the number of cases where a solver returns an "unknown" verdict, indicating that it cannot produce a counterexample or prove a property unsolvable.

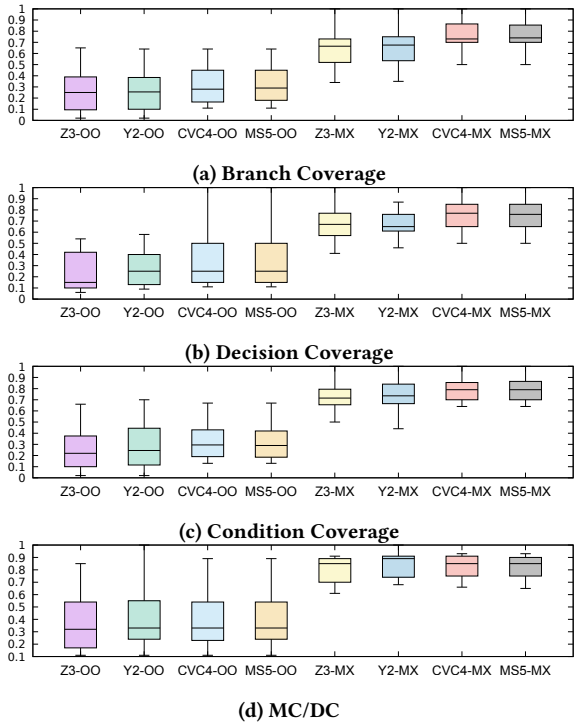
Table 2 indicates the percentage of solvable testing goals covered in situations where a solver could not achieve full satisfaction of all solvable goals for that model and criterion. From this, we can see that Z3 and Yices2 achieve full satisfaction over all 37 models, while CVC4 and MathSat5 fall short for a subset of criteria on seven of the models. For many of these models, the performance gap is not major, with CVC4 and MathSAT5 achieving over 90% satisfaction. However, for some models, like *DockingApproach* and *HysteresisAll*, the gap is much larger. We also see that more goals are missed for the more complex criteria, Condition Coverage and MC/DC.

An "unkown" verdict is generally returned for one of two reasons. First, the model could contain features that the solvers lack theories for, such as complex non-linear arithmetic operations. In this study, none of the solvers fail for that reason. Instead, the reason for failure is that the solver is unable to prove or disprove the property in the specified time limit or path length.

In this experiment, we provided all solvers with the same timeout, either 72000 seconds or a path with a length of over 500 transitions. Both limits are well over what is generally required. In practice, almost all test cases for these criteria have paths of fewer than five transitions, and test goals can be satisfied in a matter of seconds. A time out often indicates that the solver will never be able to produce an answer for that obligation, regardless of the limit. This is generally due to the model having a large and complex state space that the solver is unable to explore. In such cases, it seems that Z3 and Yices2 are better able to navigate the state spaces of these seven models, while CVC4 and MathSAT5 become trapped.

Table 3: Median percentage of mutants detected for all models where solvers achieved full goal satisfaction, and for a filtered subset of models where there are performance differences. Results are shown for all oracle strategies and coverage criteria.

			Output-Only Oracle (OO)				Maximum Oracle (MX)			
	Criterion	**Num. Models**	**Z3**	**Yices2**	**CVC4**	**MathSAT5**	**Z3**	**Yices2**	**CVC4**	**MathSAT5**
All Models	**Branch**	28	46.34%	42.30%	56.85%	56.85%	76.77%	77.36%	93.41%	93.71%
	Decision	30	77.15%	78.94%	78.14%	78.24%	100.00%	100.00%	100.00%	100.00%
	Condition	30	65.55%	84.88%	66.93%	66.93%	93.67%	100.00%	94.95%	94.95%
	MC/DC	30	75.43%	94.29%	77.88%	77.88%	95.37%	100.00%	96.37%	96.51%
Differences	**Branch**	16	25.00%	25.55%	27.90%	28.94%	66.72%	67.08%	73.03%	74.27%
	Decision	15	14.57%	25.00%	25.00%	25.00%	66.95%	65.25%	76.81%	76.77%
	Condition	16	21.77%	24.43%	29.62%	28.81%	71.51%	73.39%	78.75%	78.85%
	MC/DC	15	31.93%	32.77%	32.77%	32.77%	84.67%	88.57%	85.43%	84.85%



(a) Branch Coverage

(b) Decision Coverage

(c) Condition Coverage

(d) MC/DC

Figure 1: Boxplots of percentage of mutants detected for each solver and oracle, separated by criterion. OO = output-only, MX = maximum, Y2 = Yices2, MS5 = MathSAT5.

> CVC4 and MathSAT5 fail to produce test cases for a subset of the goals on seven models, while Z3 and Yices2 achieve satisfaction over the full set.

From this, we can see that—for the majority of our models—a tester could reasonably choose between all four solvers. However, solvers are also not a "solved" problem. More research is clearly needed before all solvers can handle all models equally well.

4.2 Fault-Detection Capabilities

Our remaining research questions examine the impact of solver choice on the fault-detection capabilities of generated test suites. To fairly assess such potential, we remove from consideration the seven models where CVC4 and MathSat5 fail to achieve full satisfaction. Failing to produce test cases will generally negatively impact fault-detection and bias comparison. We instead compare results for the remaining 30 models, where all four solvers achieve full satisfaction[5].

Table 4: Models where the generated test suites differ in performance. "i" = int, "b" = bool. Bold indicates that performance differed for all coverage criteria.

Model	# Inputs	# Internal Variables	# Outputs
CarAll	2 (b)	8 (3 i, 5 b)	1 (b)
cd	1 (i)	6 (2 i, 4 b)	1 (b)
DragonAll	13 (1 i, 12b)	22 (7 i, 15 b)	1 (b)
DragonAll2	13 (1 i, 12b)	27 (11 i, 16 b)	1 (b)
durationThm1	5 (2 i, 3 b)	7 (5 i, 2 b)	1 (b)
ex3	2 (b)	5 (2 i, 3 b)	1 (b)
ex8	2 (b)	5 (2 i, 3 b)	1 (b)
fast_1	14 (1 i, 13 b)	19 (b)	1 (b)
FireFly	9 (1 i, 8 b)	17 (7 i, 10b)	1 (b)
IllinoisAll	10 (1 i, 9b)	16 (5 i, 11b)	1 (b)
Infusion_Manager	20 (14 i, 6 b)	861 (797 i, 64 b)	5 (4 i, 1 b)
MoesiAll	5 (1 i, 4 b)	12 (5 i, 7 b)	1 (b)
ProducerConsumerAll	4 (1 i, 3 b)	12 (6 i, 6 b)	1 (b)
Speed2	2 (b)	5 (3 i, 2 b)	1 (b)
SteamBoilerNoArr2	19 (16 i, 3 b)	3 (2 i, 1 b)	1 (b)
Swimmingpool1	8 (2 i, 6 b)	21 (13 i, 8 b)	1 (b)
SynapseAll	4 (1 i, 3 b)	10 (5 i, 5 b)	1 (b)
Traffic	1 (i)	3 (2 i, 1 b)	1 (b)

Table 3 lists the median percentage of mutants detected by generated test suites for each combination of solver, oracle strategy, and criterion. First, we list the median for the full set of thirty models. From this, we can see that there are performance differences between the solvers. In fact, the difference in median is often quite pronounced—for example, in Branch, Condition, and MC/DC Coverage with the OO oracle. From first glance, it seems that there are substantive differences between solvers.

Looking at performance across the full set of models does not paint a completely clear portrait of performance. For twelve of the thirty models, every solver produces tests that perform identically in fault detection. It is worth closely analyzing performance for the models where the solvers *differ* in performance. Table 4 lists the 18 models where performance differed for at least one coverage criterion. The models in bold differ in performance across all four

[5]We exclude two models, *6counter* and *UMS* for Branch Coverage, as these two models have no testing goals for that criterion.

Table 5: P-Values for Mann-Whitney rank-sum tests.

OO Oracle

	Z3	Yices2	CVC4	MathSAT5
Z3	-	0.83	0.99	0.98
Yices2	0.17	-	0.85	0.85
CVC4	**0.01**	0.15	-	0.49
MathSAT5	**0.01**	0.15	0.51	-

MX Oracle

	Z3	Yices2	CVC4	MathSAT5
Z3	-	0.69	1.00	1.00
Yices2	0.31	-	0.99	0.99
CVC4	**< 0.01**	**0.01**	-	0.49
MathSAT5	**< 0.01**	**0.01**	0.52	-

coverage criteria. In Table 3, we also list the median performance for the subset of models where performance differs between solvers for that criteria, along with the number of models in that subset. In Figure 1, we show fault detection for the filtered set of models.

> The solvers yield test suites that differ in fault detection for eighteen models. For thirteen models, performance differences are seen across all coverage criteria.

From the filtered results in Table 3 and Figure 1, we can see clear differences in performance between the test suites produced by the four solvers. For Branch, Condition, and Decision Coverage, CVC4 and MathSAT5 have higher median performance—and higher first and third quartile performance—than Yices2 and Z3. For MC/DC, however, Yices2 ties on median performance with CVC4 and MathSAT5 for the OO oracle and pulls ahead in median on the MX oracle. Across the board, Z3 yields the worst performance, with lower medians and lower first quartiles.

> For Branch, Decision, and Condition Coverage, when solvers yield differing results, CVC4 and MathSAT5 outperform other solvers in mutant detection by up to 71.58% overall, or up to 1270% per model.

> For MC/DC, Yices2 ties on median performance with CVC4/MathSAT5 for the OO oracle and yields up to a 4.6% overall improvement for the MX oracle (up to 238.98% improvement on a per-model basis).

We perform statistical analysis to assess our observations. For each pair of solvers, we formulate hypothesis and null hypothesis:

- H: Test suites generated using solver A will have a different distribution of fault detection results than suites generated using solver B.
- $H0$: Observations of fault detection results for both solvers are drawn from the same distribution.

Our observations are drawn from an unknown distribution; To evaluate the null hypothesis without any assumptions on distribution, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [47], a non-parametric test for determining if one set of observations is drawn from a different distribution that another set. We apply the test for each pairing of techniques and baselines with $\alpha = 0.05$. Because of the limited number of samples per coverage criterion, we combine observations for all four criteria. We assess hypotheses for systems where solvers attain differing results.

P-values are listed in Table 5. The results add further weight to our previous observations. For the OO oracle, we are able to reject the null hypothesis for CVC4 and MathSAT5 versus Z3. For the MX oracle, we are also able to reject the null hypothesis for CVC4 and MathSAT5 versus Yices2. We are unable to reject the null hypothesis in all other cases.

> When using a standard output-based oracle, suites generated by CVC4 and MathSAT5 outperform suites generated by Z3 with statistical significance. When using a maximal oracle, CVC4 and MathSAT5 outperform Yices2 with significance.

4.3 Factors That Influence Fault Detection

Our goal is not necessarily to recommend one particular solver. We would not use these results to suggest the universal application of CVC4 or Yices2—particularly as these two solvers fail to achieve full goal satisfaction over several models. Rather, we want to understand *why* performance differs, so that we can make recommendations for solver improvement across the board. To that end, we have examined the impact of test oracle employed, the source of testing goals, and the data types of input variables.

Choice of Test Oracle:

In this experiment, we used two test oracle formulations. The output-only oracle reflects standard practice, where we specify expected values for the output variables of the model. The maximum oracle reflects a situation where we specify expected values for all model variables, internal or output. This is not a realistic oracle. However, it is useful for looking at the "best-case" performance of these test suites, where all triggered faults are also detected. Therefore, the maximum oracle is a useful piece of information for understanding a critical feature of solver performance—the ability of the solver to choose input that both triggers faults **and** propagates corrupted state to the model output.

The same overall trends—i.e., CVC4 and MathSAT5 outperforming Z3—largely hold across both oracles. However, there are a few observations we can make. First, there is a clear performance gap between the two oracle types. This indicates that some faults are triggered, but the corrupted state is not propagated to the output level. Intermediate calculations *mask* out the corrupted state—for example, a corrupted variable may be used in a Boolean expression, but not impact the outcome of that expression. This does not mean these faults are unimportant. Rather, it means that the chosen input was not sufficient to reveal the fault [21, 24, 38].

We can see from the results in Figure 1 that the overall distribution of results is "narrower" for the MX oracle than the OO oracle for each technique—there is less distance between the first and third quartile. This suggests that there is more variance in whether faults are propagated to the output than there is in whether faults are triggered for *all four* solvers. No solver has a universally narrower range of results, this varies by coverage criterion. Therefore, rather than suggesting a difference between these four solvers, the test oracle results identify an area of improvement for both future solvers and test generators:

Solvers and test generation frameworks could improve fault detection by factoring effect propagation of important variables into input selection.

To some extent, propagation is a factor that can be addressed through the design of coverage criterion. For example, the criteria used as testing goals in this study have been enhanced in the past with "observability" requirements that add path conditions to their existing goals [38, 46].

Ensuring propagation is clearly important, and could be addressed at one of two levels. Either the solver could ensure effect propagation by explicitly factoring path constraints into the constraints being solved, or the test generation framework could append path constraints to the test obligations being passed to the solver. In either case, these path constraints would "protect" variables referenced in the user-specified constraints from being masked along the path between their assignment and the assignment of output variables.

Choice of Coverage Criterion:

All four of the sources of testing goals used in this study are related—they are structural coverage criteria based on Boolean expressions. Therefore, we would not expect major differences in fault detection based solely on the choice of criterion. Indeed, Table 3 and Figure 1 indicate similar trends between Branch, Coverage, and Decision Coverage in terms of relative solver performance. Coverage criterion does influence the resulting fault detection—MC/DC-satisfying suites tend to detect more faults than Decision-satisfying suites—but that rise occurs regardless of the chosen solver.

However, from the results for MC/DC, we can also see that there is some relation between solver performance and source of testing goals. For Branch, Condition, and Decision Coverage, there are clear median and distribution differences between solvers. For MC/DC, both the median performance and the overall distributions (first/third quartiles in the boxplots) are quite similar across solvers.

MC/DC is the strictest of the studied criteria. It imposes the most constraints on the test input used to satisfy its goals. It follows that the solver would have less freedom to choose input that meets the goals of MC/DC. In turn, this explains why the solvers yield similar—albeit not identical—performance on models when MC/DC is the goal. The other criteria place fewer constraints. A wider variety of input will be able to attain satisfaction of those criteria. In turn, this leads to more variance between solvers.

Complex goals constrain the solver's ability to choose input, narrowing the performance gap. Simple goals can be satisfied by a wider range of input, magnifying solver differences.

The choice of testing goal is ultimately up to the developer. However, as solvers increase in efficiency and ability to solve complex constraints, it becomes easier for developers to make use of stronger coverage criteria as their baseline approach.

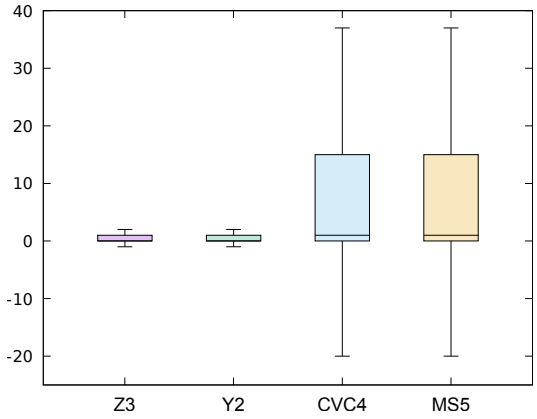


Figure 2: Values chosen for the integer input for *DragonAll2*. OO = output-only oracle, MX = maximum oracle, Y2 = Yices2, MS5 = MathSAT5.

Input Data Types:

Like in standard programs, model variables have data types—i.e., Boolean, integer, real valued, and enumerated types[6]. Solvers have evolved over the years to better handle complex data types. Some constraints remain challenging to solve—such as non-linear numeric operations—and some will likely remain unsolvable. However, great strides have been made in handling complex properties. How these data types are handled, however, differs greatly between solvers, and much of the performance gap may come down to how solvers explore the underlying structure of the studied models.

As indicated in Table 1, the studied models use a mixture of Boolean and integer input. Table 4 lists the models where there were performance differences between solvers. The models in bold are particularly important, as these are the models where differences were encountered regardless of the choice of coverage criterion. This indicates situations where the difference in search strategy clearly impacted the results.

One factor we can see, comparing the full list of models in Table 1 to the filtered list in Table 4 is that the models where test suites differ in performance tend to have numeric input variables. In contrast, many of the models where there were no performance differences only have Boolean input. This makes intuitive sense—Boolean variables only have two choices, while numeric variables can be assigned a far wider range of values. In situations where a model has numeric input, the differences in search strategy could be magnified.

To better understand the impact of the search strategy of each solver, we examined the values chosen for these input variables. We could see a recurring trend—in many cases, CVC4 and MathSAT5 choose a far wider range of values for the numeric variables.

An example of this is shown in Figure 2. The model visualized, *DragonAll2* has one integer input. All four solvers supply values for this variable as part of the counterexample, but they have different tendencies in how they set this value. All four have similar medians—Z3 and Yices2 have a median of 0, CVC4 and MathSAT5 have a

[6]Note—while other data types are supported by solvers, the models in the benchmark we used only used integer and boolean variables.

median of 1. However, as can be seen in the boxplot, CVC4 and MathSAT5 apply a much larger range of values, and apply values outside the median more often, than the other solvers.

Past research on test generation has shown that more faults tend to be detected as either the test suite grows [31] or when random assignment is used instead of a constant, default value [24]. In both cases, the reason is the same. By applying a wider range of values, the test suite is more likely to stumble on values that expose the underlying fault. Because CVC4 and MathSAT5 try a wide range of integer input, they produce suites that detect more faults—suites produced by MathSAT5 for *DragonAll2* outperform suites produced by Z3 or Yices2 by up to 360.07%.

Not all of the models with differing performance have integer input. For example, *ex*3 only has Boolean input. We examined the test suites produced for this model, and observed a similar phenomenon. This was one of the few models where Z3 outperforms CVC4 and MathSAT5—albeit only by a small amount, 3.63% when targeting Branch Coverage. When examining the suites, we saw that Z3 sets Boolean variables to `true` more often than the other solvers. The effect is more muted than with numeric variables, but by trying a wider range of values, Z3 generates better test suites.

> Solvers could improve fault-detection by applying a wider variety of values to variables, rather than tending towards fixed values.

In Section 2, we noted that the counterexamples produced by solvers are a *partial* test specification. The solver will only provide values for variables that directly impact the taken execution path. In many situations, the solver will not include values for all input variables. This is a deliberate design decision that offers benefits in terms of both computational complexity—solvers can simplify the state space by dropping unnecessary variables—and in terms of aiding human understanding—counterexamples with fewer variables are easier to read and comprehend.

If a value is not provided in the counterexample, the test generation framework itself must fill something in. This will often be a default value, like "0" for numeric variables or `false` for Boolean variables. As shown above, constantly using the same value tends to decrease the probability of exposing faults. In practice, then, there are two reasons why the tests might tend towards those values. Either the solver deliberately uses those values, or by omitting a variable, the test generation framework fills in those values.

Both of these reasons contribute to loss in fault-detection potential. In the above example, *DragonAll2*, Z3 and Yices2 explicitly set the value of the integer input. The solvers naturally tend towards a limited range of integer input. However, in other cases like *durationThm1*, Z3 and Yices2 ignore this variable—allowing the test generation framework to fill in a default "0" value. CVC4 and MathSAT5 do apply values to the input, and they detect more faults as a result.

Examining all of the produced counterexamples, we found that Z3 and Yices2 assign values to an average of 68.67% of the input variables, CVC4 assigns 70.75% of the variables on average, and MathSAT5 sets 73.10% of the variables on average. The increase for CVC4 and MathSAT5 may partially contribute to their superior performance—they have more opportunities to vary the values of those variables. Across the board, however, we can see from these averages that no solver sets values to all of the inputs for many of these models.

The reasons that solvers omit variables are valid, and we would not suggest that solvers must change their design to explicitly assign values to all variables to improve test generation—only one of the many applications of solver technology. Instead, this is an area where the test generation framework could step in to improve performance. Better strategies than a simple application of a single default value could be used to build on the gaps in the core specification offered by the solver.

> Solvers, by design, omit unused input variables from the counterexample. The strategy used by the test generation framework to fill those gaps impacts fault detection.

Parameter Tuning:

To avoid bias in our experiments, we used the same settings for the JKind model checker, regardless of the solver employed. However, each solver offers a number of parameters that can be used to tune the search. As has been seen time and time again, general solutions can be outperformed by solutions tuned to a particular problem [32]. This is certainly true in this domain as well. By tuning the parameters used by the solver to control it search strategy, we may see vast improvements in performance.

However, a developer interested in generating test cases should never have to spend extensive time tuning low-level parameters to attain a reasonable set of test cases. It would be better to ensure that test generation attempts yield "good enough" performance out of the box. This does not, however, imply that we should just use the default parameter values for a solver. Instead, we can propose an improvement for the test case generation.

Automated approaches to parameter tuning have been proposed in multiple domains, generally making use of some form of reinforcement learning [3, 32]. A model-based test generation framework could perform a short parameter tuning experiment before generating a test suite. During this experiment, it could generate a set of mutants, formulate a small set of test obligations, and generate input satisfying those obligations. It could then assess success using the set of mutants. After experimenting with various configurations, the parameters that yield the most effective mutant detection could be used to generate the final test suite. This process would require a longer test generation period, but may help yield more effective test suites.

> Solvers often offer parameters that can be used to control the search. These parameters could be automatically tuned by test generation frameworks.

5 THREATS TO VALIDITY

External Validity: Our study has focused on a relatively small number of models. Nevertheless, we believe the range of systems chosen is appropriate. The models vary in size and have been used

in prior research [22, 24, 38, 39]. Further, only models written in the Lustre language have been used. Lustre is used as it is a common intermediate language that visual models can be translated into, often before further translation into C [23]. Lustre can be straightforwardly translated into other modeling languages. Only four solvers have been used in this study. Others exist. However, these are the only four compatible with the bounded model checking framework used in this study. All four have been used extensively in research, and represent a range of strategies and underlying algorithms.

Construct Validity: In our study, we measure fault finding over seeded faults, rather than real faults. However, Andrews et al. showed that seeded faults lead to similar conclusions to those obtained using real faults [5] for the purpose of measuring test effectiveness and Just et al. found a positive correlation between mutant detection and fault detection [33]. We have assumed these conclusions hold true in our domain and language, where examples of real faults are rare.

To control experiment costs, we limited the number of mutants to 500 per model. When more than 500 mutations exist, a random selection was used to avoid bias. While the selection of specific mutants is randomized, the distribution is matched to the full distribution of possible mutants in the model. In our experience, sets containing more than 100 mutants result in similar fault finding; we generated up to 500 to further increase our confidence that no bias was introduced.

Lau and Yu [35] and Kaminski et al. [34] have defined fault hierarchies for Boolean expressions, outlining cases where detection of one fault could guarantee detection of other, redundant faults. The mutation operators we have employed could produce redundant mutations. We do not identify and remove these. However, we have performed a worst-case analysis, and found that there is a low correlation between the percent of redundant faults and the percent of detected faults. We do not believe that redundancies have a significant impact on our results.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions beyond these analyses are met, and have favored non-parametric methods. In cases in which the base assumptions are clearly not met, we have avoided using statistical methods.

6 RELATED WORK

Solvers are an integral part of model-based test generation [45], particularly methods based on bounded model checking [29, 40] and constraint solving [11, 12]. Solvers are also an important part of test generation based on symbolic execution [49], where they used to produce test input targeting structural elements. To our knowledge, this is the first study to contrast the use of solvers in terms of their ability to produce fault-exposing test suites.

This study is comparable to other studies on *parameter tuning* [1]. Many algorithms have a variety of configurable parameters—i.e., the choice of solver. We may not know a priori which choice is best, and configuration performance may depend on the problem or domain being studied [18]. Parameter tuning studies attempt to identify, either manually or automatically, the ideal settings for particular tasks [1, 16, 18]. Parameter tuning is common in test generation, particularly in search-based test generation—test generation using metaheuristic optimization techniques [37]—as evolutionary algorithms have a large number of configurable parameters that impact effectiveness [6]. Parameter tuning has also been used in model-based test generation for choices other than the choice of solver [2, 28]. Our study could be considered a manual exploration of the parameter space for solver choice.

7 CONCLUSIONS

A common form of model-based test generation is based on the use of bounded model checking, where properties expected of the model are embedded as temporal logic formulae. In bounded model checking, SMT solvers explore the state-space of the model in search of states that violate the specified properties. If the model checker finds that a predicate can be violated, it produces a counterexample—a partial test specification demonstrating the violation. The choice of solvers is important, as each follows their own algorithm, and each may produce differing counterexamples. In practice, the choice of solver could have an impact on the ability of the generated test suite to cause failures in the model or the system. To help developers choose the right solver, we must understand the differences in the test cases produced by those solvers.

We have performed experiments examining the impact of solver choice across multiple dimensions, examining the ability to attain goal satisfaction and fault detection when satisfaction is achieved—varying the source of test goals, data types of model input, and test oracle. The results of our experiment show that solvers vary in their ability to produce counterexamples, and—for models where all solvers achieve goal satisfaction—in the resulting fault detection of the generated test suites. The choice of solver has an impact on the resulting test suite, regardless of the oracle, model structure, or source of testing goals.

The results of this study identify factors that impact fault-detection effectiveness. Solvers and generation frameworks could improve fault detection by factoring effect propagation of important variables, like those associated with testing goals, into input selection. Solvers can produce improve fault-detection by applying a wider variety of values to variables, rather than tending towards fixed values. Solvers, by design, omit unused input variables from the counterexample. The strategy used by the test generation framework to fill those gaps could improve fault detection. Finally, test generation frameworks could perform short parameter tuning exercises to make more effective use of solvers. This advice could be used to improve future solvers and future approaches to model-based test generation as well as other test generation methods that make use of solvers, like symbolic execution [49].

REFERENCES

[1] Aman Aggarwal and Hari Singh. 2005. Optimization of machining techniques — A retrospective and literature review. *Sadhana* 30, 6 (01 Dec 2005), 699–711. https://doi.org/10.1007/BF02716704

[2] S. Aljahdali and A. F. Sheta. 2010. Software effort estimation by tuning COOCMO model parameters using differential evolution. In *ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010*. 1–6. https://doi.org/10.1109/AICCSA.2010.5586985

[3] Hussein Almulla and Gregory Gay. 2020. Learning How to Search: Generating Exception-Triggering Tests Through Adaptive Fitness Function Selection. In *13th IEEE International Conference on Software Testing, Validation and Verification (ICST 2020)*. IEEE, 12.

[4] T. Alsinet, F. Manya, and J. Planes. 2003. Improved Branch and Bound Algorithms for MAX-SAT. In *Proceedings of the Sixth Int'l Conf. on the Theory and Applications of Satisfiability Testing*.

[5] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Transactions on Software Engineering* 32, 8 (aug. 2006), 608 –624. https://doi.org/10.1109/TSE.2006.83

[6] Andrea Arcuri and Gordon Fraser. 2011. On Parameter Tuning in Search Based Software Engineering. In *Search Based Software Engineering*, Myra B. Cohen and Mel Ó Cinnéide (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 33–47.

[7] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177. http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf Snowbird, Utah.

[8] A. Biere, M. Heule, H. van Maaren, and T. Walsh. 2009. *Handbook of Satisfiability: Volume 185, Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands.

[9] J. Chilenski. 2001. *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. Technical Report DOT/FAA/AR-01/18. Office of Aviation Research, Washington, D.C.

[10] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proceedings of TACAS (LNCS)*, Nir Piterman and Scott Smolka (Eds.), Vol. 7795. Springer.

[11] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. 2002. STG: A Symbolic Test Generation Tool. In *Tools and Algorithms for the Construction and Analysis of Systems*, Joost-Pieter Katoen and Perdita Stevens (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 470–475.

[12] Severine Colin, Bruno Legeard, and Fabien Peureux. 2004. Preamble computation in automated test case generation using constraint logic programming. *Software Testing, Verification and Reliability* 14, 3 (2004), 213–235. https://doi.org/10.1002/stvr.300 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.300

[13] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (July 1962), 394–397. https://doi.org/10.1145/368273.368557

[14] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[15] Bruno Dutertre. 2014. Yices 2.2. In *Computer-Aided Verification (CAV'2014) (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 737–744.

[16] A.E. Eiben and S.K. Smit. 2011. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation* 1, 1 (2011), 19 – 31. https://doi.org/10.1016/j.swevo.2011.02.001

[17] Esterel-Technologies. 2004. SCADE Suite Product Description. http://www.esterel-technologies.com/v2/ scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html.

[18] E. M. Fredericks. 2018. An Empirical Analysis of the Mutation Operator for Run-Time Adaptive Testing in Self-Adaptive Systems. In *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*. 59–66.

[19] Andrew Gacek. 2015. JKind - a Java implementation of the KIND model checker. https://github.com/agacek.

[20] A. Gargantini and C. Heitmeyer. 1999. Using Model Checking to Generate Tests from Requirements Specifications. *Software Engineering Notes* 24, 6 (November 1999), 146–162.

[21] Gregory Gay, Ajitha Rajan, Matt Staats, Michael Whalen, and Mats P. E. Heimdahl. 2016. The Effect of Program and Model Structure on the Effectiveness of MC/DC Test Adequacy Coverage. *ACM Trans. Softw. Eng. Methodol.* 25, 3, Article 25 (July 2016), 34 pages. https://doi.org/10.1145/2934672

[22] G. Gay, S. Rayadurgam, and M. P. E. Heimdahl. 2017. Automated Steering of Model-Based Test Oracles to Admit Real Program Behaviors. *IEEE Transactions on Software Engineering* 43, 6 (June 2017), 531–555. https://doi.org/10.1109/TSE.2016.2615311

[23] G. Gay, M. Staats, M. Whalen, and M. Heimdahl. 2015. Automated Oracle Data Selection Support. *Software Engineering, IEEE Transactions on* PP, 99 (2015), 1–1. https://doi.org/10.1109/TSE.2015.2436920

[24] G. Gay, M. Staats, M. Whalen, and M.P.E. Heimdahl. 2015. The Risks of Coverage-Directed Test Case Generation. *Software Engineering, IEEE Transactions on* PP, 99 (2015). https://doi.org/10.1109/TSE.2015.2421011

[25] Carlos A. González, Mojtaba Varmazyar, Shiva Nejati, Lionel C. Briand, and Yago Isasi. 2018. Enabling Model Testing of Cyber-Physical Systems. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*. ACM, New York, NY, USA, 176–186. https://doi.org/10.1145/3239372.3239409

[26] G. Hagen. 2008. *Verifying safety properties of Lustre programs: an SMT-based approach*. Ph.D. Dissertation. University of Iowa.

[27] N. Halbwachs. 1993. *Synchronous Programming of Reactive Systems*. Klower Academic Press.

[28] M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. 2014. Search Based Software Engineering for Software Product Line Engineering: A Survey and Directions for Future Work. In *Proceedings of the 18th International Software Product Line Conference - Volume 1 (SPLC '14)*. ACM, New York, NY, USA, 5–18. https://doi.org/10.1145/2648511.2648513

[29] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. 2002. A Temporal Logic Based Theory of Test Coverage and Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Joost-Pieter Katoen and Perdita Stevens (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 327–341.

[30] M. Huth and M. Ryan. 2006. *Logic in Computer Science: Modeling and Reasoning about Systems, Second Edition*. Cambridge Press.

[31] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

[32] Yue Jia, Myra B. Cohen, Mark Harman, and Justyna Petke. 2015. Learning Combinatorial Interaction Test Generation Strategies Using Hyperheuristic Search. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 540–550. http://dl.acm.org/citation.cfm?id=2818754.2818821

[33] 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 *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*. Hong Kong, 654–665.

[34] Gary Kaminski, Paul Ammann, and Jeff Offutt. 2013. Improving logic-based testing. *Journal of Systems and Software* 86, 8 (2013), 2002 – 2012. https://doi.org/10.1016/j.jss.2012.08.024

[35] Man F. Lau and Yuen T. Yu. 2005. An Extended Fault Class Hierarchy for Specification-based Testing. *ACM Trans. Softw. Eng. Methodol.* 14, 3 (July 2005), 247–276. https://doi.org/10.1145/1072997.1072998

[36] D. Lee and M. Yannakakis. 1996. Principles and methods of testing finite state machines-a survey. *Proc. IEEE* 84, 8 (1996), 1090–1123. https://doi.org/10.1109/5.533956

[37] Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* 14 (2004), 105–156.

[38] Y. Meng, G. Gay, and M. Whalen. 2018. Ensuring the Observability of Structural Test Obligations. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2869146 Available at http://greggay.com/pdf/18omcdc.pdf.

[39] Anitha Murugesan, Oleg Sokolsky, Sanjai Rayadurgam, Michael Whalen, Mats Heimdahl, and Insup Lee. 2014. Linking abstract analysis to concrete design: A hierarchical approach to verify medical CPS safety. *2014 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)* 0 (2014), 139–150. https://doi.org/10.1109/ICCPS.2014.6843718

[40] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. [n.d.]. Generating test data from state-based specifications. *Software Testing, Verification and Reliability* 13, 1 ([n. d.]), 25–53. https://doi.org/10.1002/stvr.264 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.264

[41] M. Qasem. [n.d.]. SAT and MAX-SAT for the Lay-Researcher. ([n. d.]). Available at http://www.mqasem.net/sat/sat/index.php.

[42] A. Rajan, M. Whalen, M. Staats, and M.P. Heimdahl. 2008. Requirements Coverage as an Adequacy Measure for Conformance Testing. (2008), 86–104.

[43] S. Rayadurgam and M.P.E. Heimdahl. 2001. Coverage Based Test-Case Generation Using Model Checkers. In *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*. IEEE Computer Society, 83–91.

[44] RTCA/DO-178C. [n.d.]. Software Considerations in Airborne Systems and Equipment Certification.

[45] 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. https://doi.org/10.1002/stvr.456 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.456

[46] M. Whalen, G. Gay, D. You, M.P.E. Heimdahl, and M. Staats. 2013. Observable Modified Condition/Decision Coverage. In *Proceedings of the 2013 Int'l Conf. on Software Engineering*. ACM.

[47] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), pp. 80–83. http://www.jstor.org/stable/3001968

[48] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. 2001. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design (ICCAD '01)*. IEEE Press, Piscataway, NJ, USA, 279–285. http://dl.acm.org/citation.cfm?id=603095.603153

[49] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 114–124. https://doi.org/10.1145/2491411.2491456