Science of Computer Programming 78 (2013) 345–363

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

# Semantic mutation testing

John A. Clark [a], Haitao Dan [b], Robert M. Hierons [b,*]

[a] Department of Computer Science, University of York, York, YO10 5GH, United Kingdom
[b] School of Information Systems, Computing and Mathematics, Brunel University, Uxbridge, Middlesex, UB8 3PH, United Kingdom

ARTICLE INFO

*Article history:*
Available online 20 April 2011

*Keywords:*
Mutation testing
Semantics
Misunderstandings

ABSTRACT

Mutation testing is a powerful and flexible test technique. Traditional mutation testing makes a small change to the syntax of a description (usually a program) in order to create a mutant. A test suite is considered to be good if it distinguishes between the original description and all of the (functionally non-equivalent) mutants. These mutants can be seen as representing potential small slips and thus mutation testing aims to produce a test suite that is good at finding such slips. It has also been argued that a test suite that finds such small changes is likely to find larger changes. This paper describes a new approach to mutation testing, called semantic mutation testing. Rather than mutate the description, semantic mutation testing mutates the *semantics* of the language in which the description is written. The mutations of the semantics of the language represent possible *misunderstandings* of the description language and thus capture a different class of faults. Since the likely misunderstandings are highly context dependent, this context should be used to determine which semantic mutants should be produced. The approach is illustrated through examples with statecharts and C code. The paper also describes a semantic mutation testing tool for C and the results of experiments that investigated the nature of some semantic mutation operators for C.



## 1. Introduction

Testing is an important but expensive part of the software development process, often consisting of in the order of fifty percent of the overall development budget. Test automation has the potential to make testing more efficient and effective and thus to lead to cheaper, higher quality software.

Mutation testing is an approach to test automation that aims to produce test cases that are good at distinguishing between some description $N$ and variants of it. Each variant is produced by applying a mutation operator to $N$. A test case $t$ *kills* a mutant $M$ of $N$ if it distinguishes between $M$ and $N$, typically by $M$ and $N$ producing different output when run with $t$. A mutant $M$ of $N$ is said to be an *equivalent mutant* if no possible test case kills $M$. In mutation testing, either a test suite is judged against the mutants created (by determining what percentage of non-equivalent mutants are killed by the test suite) or a test suite is produced to kill all of the non-equivalent mutants. The motivation is that a test suite that is good at distinguishing $N$ from variants of $N$ is likely to be good at finding faults that are similar to applications of the mutation operators.

In traditional mutation testing, the mutation operators are designed to represent syntactically small errors. For example, an operator might replace $+$ by $-$ in an arithmetic expression. In this paper, we propose a different approach to mutation testing, namely semantic mutation testing (SMT). When testing an entity, test generation is based on a model written in some description language (such as a programming language, a design language or a specification language). While many mistakes are slips, other mistakes are the consequence of a misunderstanding of the semantics of the description language. Such misunderstandings may be captured by mutating the semantics of the description language. It is possible to introduce

* Corresponding author.
*E-mail addresses:* jac@cs.york.ac.uk (J.A. Clark), haitao.dan@brunel.ac.uk (H. Dan), rob.hierons@brunel.ac.uk (R.M. Hierons).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.03.011

changes that reflect *small misunderstandings* regarding the language through making small changes to the semantics of this language. Such changes result in the same model being interpreted in a different way. This contrasts with traditional mutation testing in which small changes are made to the syntax of the model. SMT thus aims to find a different class of fault and should complement traditional mutation testing. When dealing with a programming language, semantic mutation testing can be seen as a process that mutates the compiler rather than the program. Previous work has discussed semantic size in the context of mutation testing [1] but this work did not discuss semantic mutation testing.

This paper makes a number of contributions. First, it introduces semantic mutation, describes its potential role in testing, and explains how it can be implemented. It describes the error model of SMT and several scenarios in which SMT might play a significant role. Examples of semantic misunderstandings for statecharts and the C language are given. We investigate the corresponding mutation operators that reflect differences between semantics, demonstrating that SMT can uncover faults arising from these differences. The differences between SMT and traditional mutation testing are summarised. We then describe a semantic mutation tool that has been developed for C code and several semantic mutation operators that have been implemented. This is followed by results of experiments performed to investigate the nature of the implemented semantic mutation operators and how they compare with related syntactic operators. The description of the tool and the results of the experiments form the main contribution beyond the earlier conference version [2].

The paper is structured as follows. Section 2 describes traditional mutation testing and Section 3 outlines SMT. Section 4 describes the error model of SMT and several scenarios in which SMT might play a significant role. Section 5 then gives examples of situations in which SMT can be applied to statecharts and the C language. Section 6 describes a semantic mutation tool for C. Section 7 describes the experiments and their outcome while Section 8 explores the results and Section 9 discusses threats to validity. Finally Section 10 draws conclusions.

## 2. Traditional mutation testing

The idea behind mutation testing is simple and intuitively appealing. Mutants are produced by making changes to the program. These changes simulate classes of faults and test cases are produced to distinguish our original program from the mutants. A test suite distinguishing between the original program and the mutants provides confidence in it detecting such classes of faults.

Mutants are produced through the application of mutation operators. Each of these may be applied to a relevant point in a program in order to produce a mutant. The mutation operators involve small syntactic changes. For example, $+$ might be replaced by $-$, $>$ might be replaced by $\geq$, a variable in an expression may be replaced by a constant, or part of an expression may be deleted. The use of such mutation operators is usually justified by the competent programmer hypothesis, which states that competent programmers make small mistakes [3]. There is an issue here—a competent programmer might make semantically small mistakes that cannot be captured by syntactically small changes.

When considering programs, there are several notions as to what it means for a test case $t$ to distinguish between a program $N$ and a mutant $M$. Under strong mutation testing, which is the original form of mutation testing, $M$ and $N$ are distinguished if they produce a different output on $t$ [3,4]. In weak mutation testing, $M$ and $N$ are distinguished if they produce a different value for some state variable immediately after the point at which $N$ was changed [5]. Firm mutation testing generalises these by allowing the tester to choose the point at which the value of some state variable must differ [6].

The use of only a single mutation operator will often create large numbers of mutants even when the original program is quite small. For this reason, it is normal to restrict the number of mutants produced by using only first-order mutants: those that can be produced from the original program by the single application of one mutation operator. The use of first-order mutants is justified by the coupling hypothesis that states that any test suite that kills all first-order mutants will kill most higher-order mutants. Empirical studies suggest that there is some truth in the coupling hypothesis [7] though many questions still remain. The coupling hypothesis has also been validated by theoretical work [8], although this work makes many assumptions.

Mutation testing was originally applied to programs (see, for example, [9–13,3,14,4,15,16,5,17–20,7,21–26,6]) but more recently it has been applied to other forms of descriptions such as specifications (see, for example, [27–31]). This approach involves producing test cases that kill mutants of the specification, the test cases then being applied to the code. Naturally, in order to do this we need a particular type of specification language — one that can be executed, that can be simulated, or that allows some formal reasoning. In this work we want to produce mutants that are not equivalent. In contrast, some work on applying mutation testing to Communicating Sequential Processes (CSP) specifications considers properties of the specification, and whether these are preserved, and not functional equivalence [32]. Thus, a mutant is killed if it does not satisfy the property of interest. Interestingly, in this context equivalent mutants correspond to fault tolerance and thus their existence is desirable.

Mutation testing has a number of advantages. First, it allows the tester to target particular classes of faults. Should a program pass a test suite that kills all mutants, then it is clear that the non-equivalent mutants produced were not correct. This eliminates a set of faulty behaviors. It also gives us confidence in the test suite distinguishing between a correct program and a program with one of these types of faults. Second, other test criteria may be simulated using mutation testing. Consider, for example, the mutation operator that replaces a statement by a new statement that terminates execution with an error message. Then, any test suite that kills all of the non-equivalent mutants formed using this mutation operator must also provide 100% statement coverage: every reachable statement is executed during testing.

While mutation testing is powerful and flexible, it does have disadvantages. The number of mutants produced, even when considering a small program and first-order mutants, is often massive. For example, Offutt and Pan, using a standard set of 22 mutation operators and the Mothra tool, produced 951 mutants from a program that contained only 28 executable statements [25]. For this reason, researchers have introduced *selective mutation* in which a subset of the mutation operators is applied [12,33–35]. The presence of equivalent mutants often leads to a significant amount of manual effort and increases the cost of mutation testing. There has thus been work on preventing the introduction of some equivalent mutants and automatically detecting some of the equivalent mutants that are introduced [10,16,21,23,25].

Mutation operators work at the syntactic level and thus are best at representing errors that are in the form of small slips or typos. Such mutants do not aim to represent misunderstandings that relate to a small semantic mistake but that can only be implemented through large syntactic changes. This paper introduces SMT and argues that it overcomes some of these problems. In particular, it describes situations in which semantic mutation testing has the potential to lead to the use of test cases that find particular classes of faults that are quite different from those represented by syntactic mutants. In addition, in the experiments reported in Section 7, it was found that there were far fewer semantic mutants than syntactic mutants and that the sets of semantic and syntactic mutants did not subsume one another.[1]

## 3. An overview of SMT

SMT is a powerful and general concept. It requires the use of a description language with a semantics that allows manipulation along with some notion of likely misunderstandings. Alternatively, the mutations of the semantics might explore possible variance in the semantics. For example, many programming languages have elements that are implementation specific — the compiler writer is allowed to choose between certain options. By mutating the semantics to represent these different options it is possible to explore the portability of a program. Mutation operators to be applied to the semantics of a programming language could reflect alternatives regarding, for example, the precision used for floating-point numbers.

An entity in which we are interested is represented by a (syntactic) *description* (such as the source code of a program). Given a description $N$ written in a language with semantics $L$, the behavior is defined by the pair $(N, L)$. Traditional (syntactic) mutation testing mutates one part of this: the description. Thus, the application of a syntactic mutation operator is of the form $(N, L) \rightarrow (N', L)$ for some $N'$. By contrast, SMT mutates the semantics of the language and does not change the description. Thus, the application of a semantic mutation operator is of the form $(N, L) \rightarrow (N, L')$. A first-order mutant $(N, L')$ of $(N, L)$ is one produced by applying one mutation operator once to the semantics of the language.

Suppose that $(N, L)$ is mutated to get $(N, L')$. Thus $N$ has two interpretations, its meaning under $L$ and its meaning under $L'$. These will be called $N_L$ and $N_{L'}$ respectively. Given a test case $t$, $N_L(t)$ will denote the behavior produced when applying $t$ to $N$ under semantics $L$ and $N_{L'}(t)$ will denote the behavior produced when applying $t$ to $N$ under semantics $L'$. Then a test case $t$ *kills* the mutant $(N, L')$ if and only if $N_L(t) \neq N_{L'}(t)$. Further, this mutant $(N, L')$ is an equivalent mutant if for all $t$ we have that $N_L(t) = N_{L'}(t)$. Naturally, the notions of behavior, equality of behavior, and thus of killing a mutant will depend upon the language being considered. In addition, the property of a test case killing a semantic mutant depends both on the semantic mutation made and the description under test.

There are several ways of implementing semantic mutation, including the following.

1. Have a parameterisable system for interpreting a model, the parameters allowing the semantics to be mutated.
2. Express the semantics in some form that can be manipulated. One such form is a set of rewrite rules.
3. Simulate a mutation of the semantics by making changes to the syntax of the description. Note that these will often be done *throughout* the description, not just at one point.

In the mutation testing of a description $N$ in a language with semantics $L$ a set of mutation operators are applied individually to $L$ to get alternative semantics $L_1, \ldots, L_m$. The semantic mutants $(N, L_1), \ldots, (N, L_m)$ are then used in order to evaluate a test suite or to drive test generation: a test suite should kill every non-equivalent mutant in the set $\{(N, L_1), \ldots (N, L_m)\}$.

One of the benefits of semantic mutation testing is that it may lead to far fewer mutants and, as a consequence, fewer equivalent mutants. This is because a change in the semantics of the description language need only be made once[2] (assuming only first-order mutants are used). By contrast, in traditional mutation testing, given a mutation operator there is a mutant for every point in the description to which the operator may be applied. Thus, a large number of mutants may have to be compiled and executed; in semantic mutation only one compilation is necessary for each semantic mutation operator.

## 4. Scenarios for SMT

In a development software process, multiple descriptions of the underlying software may be generated in different activities. The form of description changes in this process, generally from abstract to concrete. A number of languages may be used: scenario-based models (Sequence Diagrams and Message Sequence Charts (MSCs)) may be used in the requirements phase, more formal languages such as finite state machines, Z and VDM can be applied in the specification or design phases

[1] A set $M_1$ of mutants subsumes a set $M_2$ of mutants if the test suite produced to kill the mutants in $M_1$ also kills the mutants in $M_2$.

[2] Later we discuss conditions under which this might be relaxed.

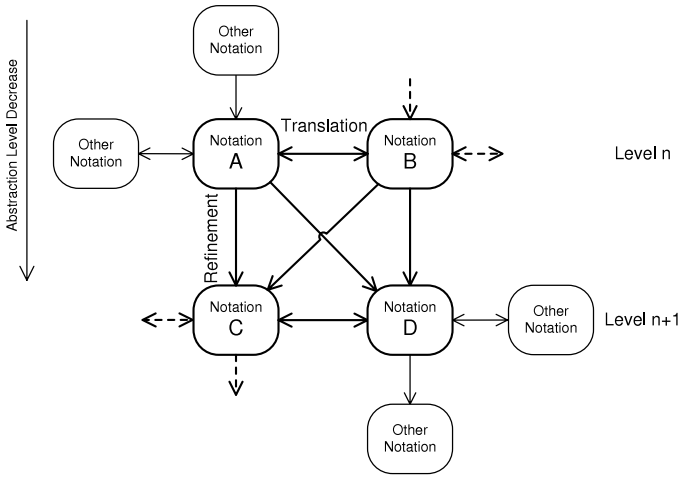


**Fig. 1.** Transformation model.

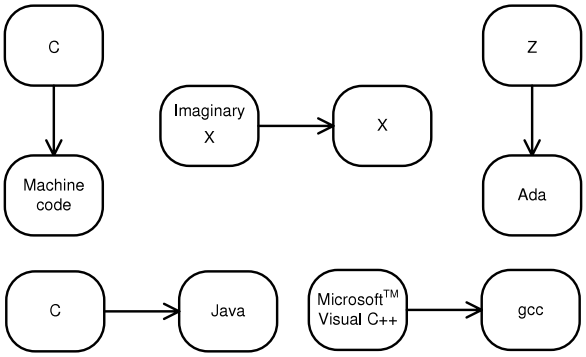


**Fig. 2.** Refinements and translations.

and finally the software may be coded in C. Semantic misunderstandings can be introduced into the target description in every transformation because of the informality of either languages or the semantic differences between the source and target languages in these transformations. In this section, we first describe a semantic error model to describe how different semantic misunderstandings can be introduced into software.

### *4.1. Semantic error model*

Fig. 1 shows a general network model of misunderstandings introduced into the final software. To simplify the illustration, we give a partial network including nodes A, B, C and D that represent four possible different descriptions of a piece of software.

In Fig. 1 the level of abstraction descends from top to bottom. Languages used in A and B or C and D are at the same abstraction level and the languages used in A and B at level $l$ are more abstract than the languages used in C and D in level $l+1$. Generally, descriptions written in higher level languages (for example, models in the design phase) will be refined into descriptions in lower level languages (source in the code phase). Therefore, downward transformation is common in software development. These types of transformations are called *refinements* and are represented by downward arrows in Fig. 1.

In addition, the description in each level may be expressed in more than one language. Description A may be translated into B (and vice versa) and Description C may be translated into D (and vice versa) in a software development process. For example, MSCs can be used to synthesise automata for model checking [36]. These types of transformations are called *translations* and are represented by left right arrows in Fig. 1.

The number of levels of abstraction may depend on the software development process used. For a simple piece of software with several hundred lines of C code, there may be only two levels: C code and compiled machine code since no formal requirements and design are needed. This scenario is shown on the left-hand side of Fig. 2. However, for a large project, there may be descriptions at many levels (requirements, specification, design, code and machine code). Similarly, the number of nodes at the same level may vary. Consider software developed in Java derived from legacy software that contains components coded in different languages. There should be multiple nodes at the code level. Therefore, in Fig. 1, we do not restrict the size of the model.

Semantic mutation is concerned with misunderstandings that arise both in refinements and translations in the network model. A specific software development process may introduce misunderstandings along a top-down path in the model and

ideally we investigate all possible misunderstandings raised by transformations. However, some of the transformations may be more interesting since misunderstandings are more likely in these scenarios. In addition, not all of the misunderstanding can be imitated. For example, if software is developed directly from informal descriptions the possible misunderstandings are hard to capture. In the rest of this section we describe several situations in which SMT may play a significant role.

### 4.2. Common misunderstandings

Given a taxonomy of common misunderstandings for a particular language, a set of semantic mutation operators could represent these possible misunderstandings. Such a set of operators could be informed by studies that identify common misunderstandings (see, for example, [37]). Given a mutation operator that represents a possible misunderstanding, a test suite that kills the mutant produced by this operator should be good at finding faults that are due to this misunderstanding. Thus, testing is targeted at these common misunderstandings.

Ideally, the set of operators used should reflect the environment in which the artifact under test has been produced and the misunderstandings that are most likely or most important within that environment. For example, it is to be expected that novice programmers make very different mistakes from expert programmers and that an expert programmer using a language for the first time will have a different set of likely misunderstandings than an expert programmer who has used that language for many years.

In the error model, if the used language is $X$ then this scenario can be described by a translation between imaginary $X$ node to $X$ node shown in Fig. 2. The imaginary $X$ node represents the programmer's understanding of language $X$. It may be noted that this scenario can happen at any level of the error model.

### 4.3. Refinement

Misunderstandings might occur through a change between the level of abstraction in requirements, specification, design and code. For example, Z [38] and Ada [39] have different truncation rules. The precedence rules may also differ between languages. Where there are similarities between elements of the syntax of a specification or design language and the programming language used there is a danger that statements written using this syntax will be copied. This may lead to faults if these constructs are given a different semantics in the specification/design language and the programming language. Semantic mutation operators could change the semantics of the programming language to simulate the semantics given to these syntactic constructs in the specification or design language. Given a set of mutants generated in this manner, a test suite that kills the resultant mutants is targeted at such mistakes.

Another example is the use of unbounded types (such as the integers) in specification languages; they are retrenched[3] to bounded types. Additional issues occur with types such as the reals since these will be retrenched to types of finite precision. The retrenchment may lead to behaviors other than those specified and the behavior may also depend upon the actual bounds and the precision. Semantic mutation operators might be used in order to explore the impact of such retrenchment and the choices regarding bounds and precision. There has been work on finding test cases to explore the effect of precision [40] and such approaches might have value in producing test cases to kill certain types of semantic mutation.

In the model of Fig. 1, these forms of semantic mutations correspond to a set of refinement transformations. For example, the truncation example can be represented by the refinement from node Z to node Ada shown in Fig. 2.

### 4.4. Migration

Let us suppose, for example, that a company uses a description notation and is migrating to a different one. The original language and the new one may encapsulate different semantics. If this is the case, there is a danger of mistakes caused by this difference in semantics. The process of migrating to the new language would be assisted by a tool that generates test cases that are good at finding mistakes caused by the change in semantics.

Let $L_0$ denote the original semantics and $L$ denote the new semantics. Let $L_1, \ldots, L_m$ denote a set of alternative semantics each of which captures a difference between $L$ and $L_0$. Given a description $N$ that has been produced for the new semantics, it would be natural to use the mutants $(N, L_1), \ldots, (N, L_m)$.

A semantic mutation tool could produce the mutants. It might then either determine which are killed by a proposed test suite or assist in a search for test cases to kill the mutants. If the tool finds a test case $t$ that distinguishes between $(N, L)$ and $(N, L_i)$ for some $0 \leq i \leq m$ then $t$, and the response of $(N, L)$ and $(N, L_i)$ to $t$, can be reported back to the developer.

An example of this scenario is migrating to a different but similar programming language. There are languages from a particular paradigm that use the same, or similar, syntactic constructs but give them different semantics. For example, C uses short-circuit evaluation while in Ada there are two versions of each logical connective, one that has short-circuit evaluation and one that does not.[4] Different languages deal with exceptions in different ways. Languages also differ in their binary representation of characters and strings and thus give a different semantics to code that directly manipulates

[3] In retrenchment we implement a type from the specification using one that does not formally conform to the original type. One major motivation is that many types in specification languages are infinite and cannot be implemented in standard programming languages.

[4] This is an example of a semantic mutation that can be easily simulated using a set of syntactic mutations.

these representations. For example, Java makes use of the 16 bit UNICODE representations while in C characters are Bytes. In Java the length of an individual string is fixed while in C there is an end of string character. Thus, code that terminates string manipulation in C, by checking for the end of string symbol, will not operate correctly in Java.

The mutation operators used will depend upon the previous and new languages/semantics. There could be suites of semantic mutation operators for common combinations: semantic mutation operator suites targeted at particular changes in semantics. Semantic mutation testing then leads to the use of test suites that are targeted towards mistakes that may result from a migration in semantics.

Within Fig. 1, migrations can happen in any level of the error model and correspond to translations in the model. For example, the migration from C to Java is shown in Fig. 2.

### 4.5. Porting of code

Many programming languages have elements of their semantics that are implementation specific. SMT can be used to explore the impact of such freedom and thus to assist in determining the portability of the code. Here, equivalent mutants represent robustness to a change in, for example, a compiler. If it is not feasible to determine whether the mutants are equivalent mutants, random testing (possibly based on a user-profile) might be used to provide confidence in there being only very limited portability issues. Specifically, randomly generated test cases can be applied to the program and semantic mutants used to represent potential portability problems and the proportion of the test cases that kill the mutants gives an estimate of the effect of potential portability problems.

Consider, for example, the order of evaluation of terms within an expression. This is not specified in C but a compiler will normally make a consistent choice. The choice made can affect the behavior of the system. To see this, consider an expression $f(x) + g(x)$. If one or more of $f$ and $g$ contains a side-effect that can affect the value of $x$ (or some shared data) then the order of evaluation is important. A simple semantic mutation operator would reverse this order of evaluation. If this mutation operator creates an equivalent mutant then the (functional) behavior of the program being tested is not affected by this portability issue. Given a programming language and a list of such issues it is possible to produce a standard set of semantic mutation operators in order to explore portability.

This semantic mutation scenario largely happens in the code level. It mainly addresses the discrepancies between two different implementations of a language, for example, *Microsoft*$^{TM}$ Visual C++ and gcc. In our error model, this scenario corresponds to translations. For example, the semantic mutation introduced by the effort to make a piece of source code that works with *Microsoft*$^{TM}$ Visual C++ also work with gcc can be modeled as the diagram on the right-hand side of Fig. 2.

## 5. Examples and characteristics of SMT

This section describes two applications of SMT: one is statecharts and the other is the C language. We thus summarise some of the characteristics of SMT. Statecharts have been chosen because there are several rival semantics encapsulated in toolsets (see, for example, [41] for more information on some of the differences).

### 5.1. Statechart: mutation for multiple semantics

Statecharts are a popular graphical notation for specifying state-based systems. Statecharts were originally introduced in order to specify reactive systems (see, for example, [42]) and are widely used in the specification of embedded control systems. They now form part of the Unified Modeling Language (UML) and are thus used to form part of the specification of object-oriented systems. The core components of a statechart specification are states and transitions between states. A transition has a label which may include any of: an event that triggers the transition, a guard, and the action of the transition. Let us suppose that $t$ is a transition from state $s$, with event $e$ and guard $g$. In order for $t$ to be triggered the system must be in state $s$, the event (or input) $e$ must be available and the guard (or precondition) $g$ must evaluate to true. In order to limit complexity, statecharts allow a state hierarchy. A state is one of: a basic state; an AND state; or an OR state. A basic state, unlike AND states and OR states, contains no other states. An AND state contains a number of substates that act in parallel: if the system is in an AND state then it is also in each of these substates. An OR state contains a number of alternative states: if the system is in an OR state then it is in exactly one of its substates.

For example, Fig. 3 shows a statechart that describes part of a simplified cruise-control system for a car. Many elements of a real cruise-control system, such as it having a maximum speed, have been left out or abstracted away in order to aid simplicity. There are two main states: in one of these the cruise-control system is active (the state `ON`), in the other it is inactive (the state `OFF`). The state `OFF` is a basic state as it has no substates while `ON` is an OR state. The car has a sensor that scans the road ahead. The state `ON` has two substates: one in which one or more vehicles have been detected (state `vehicle_in_front`) and one in which no vehicle has been detected in front (state `no_vehicle_in_front`). There is a lever for controlling the speed when the cruise-control system is in state `ON`. This lever has 3 settings: increase, null, and reduce. This lever being at setting X is denoted by `lever = X` and is seen as an event.

The intended semantics is as follows. If the system is in state `ON` then it remains in this state unless the brake is applied or it is switched off. While in state `ON`, if the system is in substate `no_vehicle_in_front` and a vehicle is detected in front then

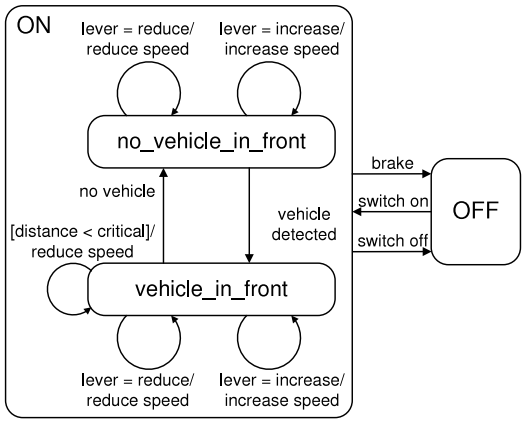


**Fig. 3.** A statechart for a cruise-control system.

the system moves to substate `vehicle_in_front`. While in state ON, if the system is in substate `vehicle_in_front` and there is no longer a vehicle detected in front then the system moves to substate `no_vehicle_in_front`. Within substates `no_vehicle_in_front` and `vehicle_in_front`, it is possible to increase the required speed and to reduce the required speed. When in state `vehicle_in_front`, the speed is reduced if the distance to the vehicle in front becomes critical.

This example will now be used to illustrate points on which two pairs of semantics (STATEMATE/UML and STATEMATE/Stateflow) differ and to show how semantic mutation testing may be used to assist in the generation of test cases to explore the impact of these differences.

### *5.1.1. A difference between STATEMATE and UML semantics*

Let us suppose that the system is in state `no_vehicle_in_front` and thus is also in state `ON`. Suppose also that the events `brake` and `lever = increase` are received. Under the STATEMATE semantic the system will move to the state `OFF` since a transition from a state takes precedence over the transitions from its substates (see, for example, [43]). This is the required behavior — the cruise-control system should not be attempting to maintain the current speed while the brakes are being applied. However, under the UML statechart semantics the transition with event `lever = increase` will initially be activated since transitions of substates take precedence over those of states containing them (see, for example, [43]). This is clearly erroneous behavior for this model.

Now let us suppose that the UML semantics are being used but it is realised that the developer might mistakenly apply the STATEMATE priority rules. Given the specification in Fig. 3, two behaviors would be analyzed:

1. The specified behavior $B$ that corresponding to the specification under the UML semantics; and
2. The behavior $B'$ resulting from mutating the UML semantics. The mutation is that the priority of transitions in different levels of the state hierarchy is changed to that used in STATEMATE.

We could then produce a test case to kill $B'$ and this would involve taking the system to the state `no_vehicle_in_front`, applying the events `brake` and `lever = increase` and observing whether acceleration occurs.

Again, let us suppose that the specification in Fig. 3 has been produced under the UML semantics. The specification is incorrect and thus testing against this specification may find that the code implements it perfectly and thus does not find a failure. Further, the fault is associated with two transitions being enabled at the same time rather than being associated with a particular transition. Thus techniques that aim to test individual transitions are unlikely to find such a fault. An example of such a technique is the transition tour method in which a test is required to traverse every transition (see, for example, [44]). By contrast, any test sequence that kills the semantic mutation produced using the operator described is guaranteed to find this fault and will suggest that a particular type of misunderstanding has occurred.

### *5.1.2. A difference between STATEMATE and Stateflow semantics*

A famous difference between the Stateflow and STATEMATE semantics for statecharts is as follows. If there is non-determinism,

1. under the STATEMATE semantics, an autocoder[5] will make a decision and thus produce a deterministic implementation [45].
2. under the Stateflow semantics, the system progresses clockwise from the upper left corner of the state and chooses the first enabled transition met [46].

[5] Autocoders automatically generate code to implement a model.

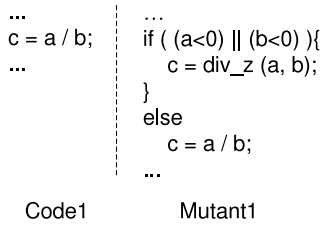


**Fig. 4.** Division with a negative number and its mutant.

Let us suppose that the system is in the state `no_vehicle_in_front`. If a vehicle is detected and the lever is at the setting increase then there are two enabled transitions: moving to state `vehicle_in_front` or increasing the speed and staying in state `no_vehicle_in_front`.

According to the STATEMATE semantics, the behavior of the system depends on the autocoder. It may use some consistent way of determining the transition chosen and define the intended behavior through giving priority to the transition from `no_vehicle_in_front` to `vehicle_in_front` over the transition from `no_vehicle_in_front` with event `lever = increase` as required. However, in this case, under the Stateflow semantics the system will fail to move to the state `vehicle_in_front` and thus will continue to behave as if there is no vehicle in front.

A mutation operator might represent this possible misunderstanding, producing a mutant that implements the possible STATEMATE semantics described above. We could then produce a test case to kill the mutant by taking the system to the state `no_vehicle_in_front`, applying the events `vehicle_in_front` and `lever = increase` and observing whether acceleration occurs.

Again, the mutant describes the potential misunderstanding: a test case kills the mutant if and only if it detects this misunderstanding. Test techniques that test individual transitions may not find such problems and so the process of investigating this problem from the perspective of SMT also reveals a weakness of such techniques.

### *5.2. C: mutation for safer C*

In this section, we briefly explore SMT with more concrete notations, and in particular, C code.

The functionality of even the 'simplest' constructs may differ between high-level notations and the C language. In the development of semantic mutation operators for C, typical or possible differences between high-level notations and C can be derived and used as the basis for generating a set of semantic mutants for refinement to distinguish such cases. In addition, the C language does not have a standard formal semantics. For example, Hatton [47] quotes 97 types of explicitly undefined functionality in the ISO C Standard [48]. The ambiguities in C semantics can be dangerous especially in porting of code. Semantic mutations can thus be designed to capture these differences.

We now give three examples to illustrate possible semantic mutations for achieving safer C programs. The first example is a possible misunderstanding in refinement from Z to C. The remaining two concern cross translations between different versions of C. The semantic mutation operators given in this section use the approach, 'simulate a mutation by making changes to the syntax of the description'.

#### *5.2.1. Division of negative numbers*

Consider division of integers. In the C language (-12/5) has the value -2, whereas the corresponding function *div* in the formal specification language Z takes the value -3. C truncates towards 0 and Z towards minus infinity.

A semantic mutation operator can be developed to modify the division expressions in C to *if* . . . *else* statements. A helper function, *div_z*(), which acts as division in Z will be used when one of the two operands of a division expression is negative. For example, *Code1* will be transformed to *Mutant1* shown in Fig. 4.

The generated mutant differs whenever truncation is applied to a negative value so if a piece of code is used in a context within which all values are positive such a semantic mutant is guaranteed to be an equivalent mutant.

#### *5.2.2. Incomplete branching structures*

In C code, *if* . . . *else* statements are used to introduce branching logic. Such a statement may not consider all cases and this can be appropriate but may also denote a mistake. For example, an incomplete *if* . . . *else* statement is given in Fig. 5. For *Code2*, if *a* is neither *b* or *c*, the program will continue without executing either of the guarded statements. This might be a semantic misunderstanding.

An incomplete logic structure may result from two types of mistakes. First, programmers may assume that the program will always execute the last branch of the structure; second, programmers may simply fail to provide a branch for the default condition. A semantic mutation operator can be developed for incomplete branching structures. It modifies the last branch to make it a default branch or inserts a default branch at the end of the branch structure. For example, *Code2* and the mutants, *Mutant2* and *Mutant2*', are shown in Fig. 5.

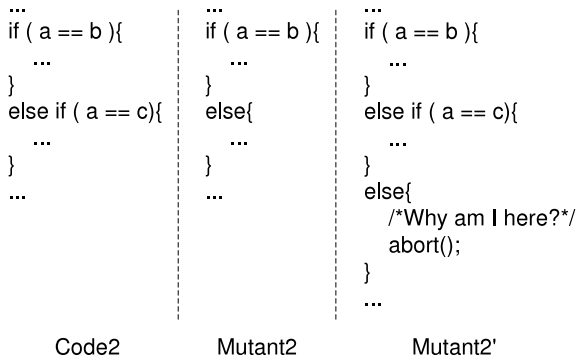


**Fig. 5.** Incomplete *if . . . else* statement and its mutants.

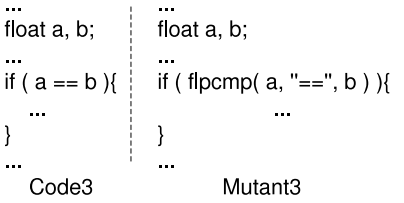


**Fig. 6.** Floating-point comparison and its mutant.

*5.2.3. Floating-point comparison*

In C code, logical comparisons between floating-point numbers are allowed. These can exist in different contexts, such as in *if* and *for* loop conditions. In addition, the logical comparisons can be equal or non-equal (bigger than or less than). For example, a floating-point comparison code snippet, Code3, is given in Fig. 6 to show the equal (if) logical comparison.

The behavior of Code3 shown in Fig. 6 is unpredictable and may differ from machine to machine since the comparison of floating-point numbers in C is not rigorously defined. A semantic mutation operator can be developed to mutate the floating-point comparison operators. To conduct the mutation, a helper function *flpcmp* is introduced, which conducts the given comparison for the two float type operands at a particular level of arithmetic granularity. In order to achieve this we use the constant FLT_EPSILON, which is typically a very small number whose value is defined in the float header file, and used this in checking whether two floating point numbers are 'close enough' to be considered to be equivalent. The value of FLT_EPSILON may vary with compilers and operating systems and this leads to changes in the arithmetic granularity.

For *Code3*, its mutant *Mutant3* is shown on the left-hand side of Fig. 6.

*5.3. Summary*

As shown in the above examples, several differences between traditional and semantic mutations can be observed.

A semantic mutation aims to simulate a misunderstanding (this is the fault or error model). The type of misunderstandings considered can depend on the context in which development is taking place. For example, if a developer has usually used one language $X$ and is now using a different language $Y$, we can use semantic mutation operators that target the types of misunderstandings that can occur when moving from $X$ to $Y$.

The process of implementing semantic mutation operators is more complex than that of implementing traditional syntactic mutation operators. This is because context is important and additional analysis may be required. For example, it is necessary to infer the types of the expressions at both sides of a relational or equality expression to implement semantic mutation operator for floating-point comparison.

Given description $N$ in language $L$, it is possible to simulate a change in the semantics of $L$ through syntactic changes to $N$. This may affect more than one construct of $N$. For example, operator for incomplete branching structures affects whole *if . . . else* statements.

A semantic mutation operator may generate fewer mutants than syntactic mutation and so there may be fewer equivalent mutants. One reason is that a semantic mutation is more specific. For example, the operator dealing with floating-point comparison only changes the relational/equality expression when one of the operands is float. Another reason is that a change in the semantics of the description language need only be made once, assuming only first-order mutants are used. By contrast, in traditional mutation testing, given a mutation operator there is a mutant for every point in the model to which the operator may be applied.

In the statechart example given, the mistakes could have been found by adding a new test objective relating to testing when combinations of transitions are enabled. There may be merit in using test criteria that achieve this. However, the purpose of this section was not to produce a test criterion for testing from statecharts; rather, it was to show how the general approach of SMT might be applied when there is the potential of misunderstandings caused by a variety of semantics.

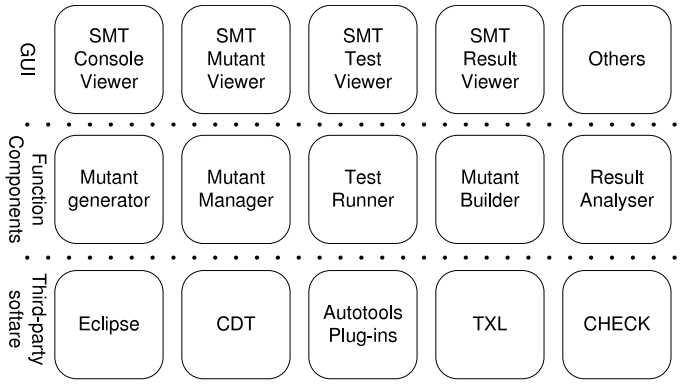


**Fig. 7.** SMT-C Architecture.

## 6. A semantic mutation tool for C

We have developed a new mutation testing tool for C, called SMT-C, because we found that no tool satisfies the requirements of SMT. In this tool, we used the approach in which semantic mutations are simulated through making changes to the syntax of the description. Our goal was to deliver a flexible and easy to use semantic mutation testing tool which can be seamlessly embedded into the daily working routines of a software engineer. In addition, we wanted a tool that could easily be extended with additional mutation operators. In this section we describe the tool's architecture, the way in which it was implemented, and the semantic mutation operators that have been developed.

### 6.1. Overall architecture

We developed SMT-C in Java and based it on the Eclipse platform. It can be run either as an independent application or a plug-in of Eclipse integrated C development environment. SMT-C has a three-layer architecture as shown in Fig. 7. The basis is third-party software, including Eclipse, TXL [49], Check [50] and others. The upper layer contains GUI components, mainly 4 viewers. In the middle layer, there are functional components, where the core features of the tool are implemented.

We now describe some main components of the tool in more detail.

### 6.2. The viewers

As shown in Fig. 7, the main GUI of SMT-C has 4 viewers: the mutant viewer, the test viewer, the results viewer and the console viewer. The upper part of Fig. 8 is the main window of SMT-C when running independently as a Rich Client Application. The mutant viewer is on the right-hand side of the main window and here the generated mutants can be managed. The results viewer is at the bottom of the main window and displays the results of testing. The test viewer and the console viewer are displayed separately below the main window. The test viewer provides a front end to the test runner and allows the tester to start the application of the test cases and view the results of each test case that has been run. The console viewer allows the tester to monitor the current status of the components that are running.

A lot of high-level features of Eclipse and CDT (C/C++ Development Tooling) have been reused in SMT-C. For example, the middle of the main window of SMT-C in Fig. 8 shows two mutants being compared and this feature is implemented reusing the compare module of Eclipse.

### 6.3. Core function components

In the middle layer, the function components of SMT-C implements functions for generating, building, testing and executing mutants. Most of these functions are developed based on third-party software in the base layer.

#### 6.3.1. The mutation generator

SMT-C implemented 13 semantic mutation operators based on the ideas in [47] which are briefly described as follows:

1. **AOR**: replace '=' with '==' in conditional statements;
   The misuse of the assignment expression in control structures is a well-known mistake in C programs.
2. **ASD**: remove additional semicolons after the condition expressions of *if* statements;
   It is possible that some programmers punctuate *if* statement as follows: *if ( a == b );{...}*. In this case, the statements in the bracket after the semicolon will always be executed.
3. **LBC_I**: add an *else* branch to the *if* statement without an *else* branch. This *else* branch contains a trap;
4. **LBM_I**: modify the last *else if* branch of an *if* statement without an *else* branch to be an *else* branch;
5. **LBC_C**: add a *default* branch to the *switch* statement without a *default* branch;

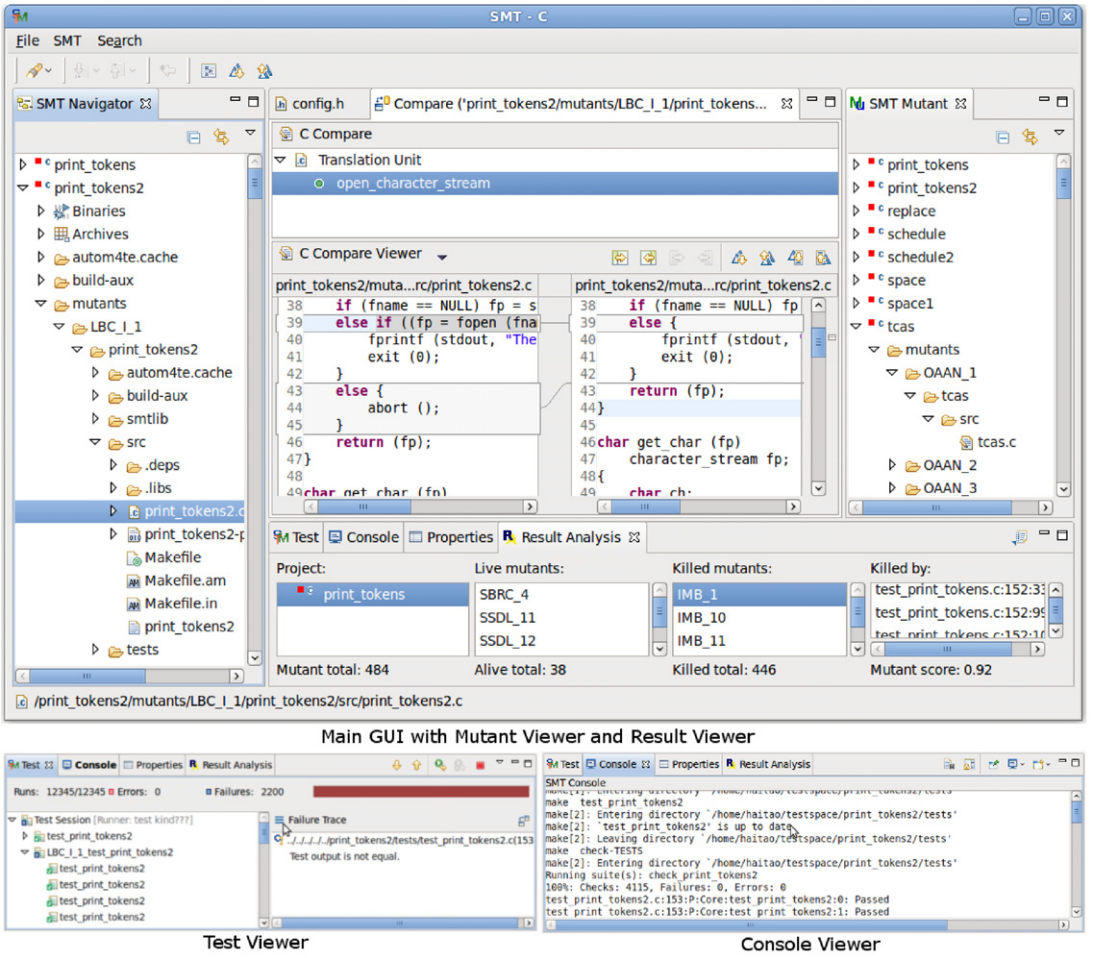


**Fig. 8.** GUI of SMT-C.

6. **LBM_C**: modify the last *case* of a *switch* statement without a *default* branch to be a *default* branch;

   **LBC_I**, **LBM_I**, and **LBC_C** and **LBM_C** are 4 operators that deal with incomplete branching structures in C programs. Incomplete branching structures can be potential errors since programmers may assume that the program will always execute the last branch of the structure and programmers may simply fail to provide a branch for default condition.
7. **MFC_E**: mutate the floating-point comparison operators in an equality expression. For example, if $a == b$ is an expression in a conditional statement in which at least one of the two variables $a$ and $b$ is a floating-point number, the MFC_E operator will change a==b to a function call $flpcmp(a, b)$ as shown in Fig. 6.
8. **MFC_R**: mutate the floating-point comparison operators in a relational expression;

   **MFC_E** and **MFC_R** are used to handle logical comparison between floating-point numbers in conditional expressions in C programs. The behaviors of comparison operators for floating-point numbers are unpredictable and may differ from machine to machine since there are no standard definitions of these operators. Programmers may underestimate the complication of comparison of floating-point numbers and generate some very subtle errors.
9. **DIA_F**: mutate the results of division/modulus of integers using the floor method;
10. **DIA_T**: mutate the results of division/modulus of integers using the tail method;

    Division of integers is interpreted differently in different programming languages. For example, in Ada the expression $(-12/5)$ evaluates to $-2$, whereas the corresponding expression in the formal specification language *Z* takes the value $-3$ [51].

    According to the C standard [52, 3.3.5], if either operand is negative, whether the result of the / operator is the largest integer less than the algebraic quotient or the smallest integer greater than the algebraic quotient is implementation-defined. Again, programmers may underestimate the situation and generate errors.
11. **FTA_F**: floating type truncation adjustment using the floor method;
12. **FTA_T**: floating type truncation adjustment using the tail method;

    When a floating-point number is assigned to an integer variable the number is truncated. However, according to [47], the direction of truncation is undefined in C++ [53] or Java [54]. In C, it is defined using the floor method.

This may lead to semantic misunderstanding since C programmers with different backgrounds may interpret C statements based on their experience with C++ or Java.

13. **IMB**: inserting missing *break* statements into *switch* statements.

    The case statement in C is not as safe as those in Ada or BASIC. A flaw of particular notoriety arises through the omission of the *break* statement at the end of each switch branch. Execution will simply 'fall through' each switch branch. Therefore, programmers with different backgrounds may ignore the need of *break* statements and introduce faults.

In addition, SMT-C also implements some related traditional mutation operators defined in [55], in part because this allows us to run experiments that compare semantic and syntactic mutation operators. The mutant generator is implemented based on TXL [49] which is a generalised source-to-source translation system. It takes as input a piece of source code, the grammar used by the source code and a set of transformation rules written in the TXL programming language and then produces the transformed source code. It is straightforward to implement most of the proposed semantic mutation operators and traditional mutation operators. For example, the logic of operator **SCRB** (break to continue) can be implemented using several lines of code as follows:

```
replace $ [jump_statement]
  'continue _ [semi]
by
  'break;
```

This piece of code essentially means: find each jump statement which is formed by the string 'continue' and a semicolon and then replace it with a new jump statement 'break;'. TXL has a powerful set of built-in functions which makes it very flexible and as a result it can be used to implement relatively complex transformations.

It transpired, however, that it was difficult to implement six of the proposed semantic mutation operators using TXL because they require us to have information regarding variable types. For example, the mutation operator, **MFC_R**, needs type information of variables in relational expressions. In order to solve this problem, we implemented a type annotation module for the C language. This led to a library with 2.5k lines of code in TXL. SMT-C uses this to parse the C source code to form a grammar tree that contains type information. This not only helped us to implement the 6 semantic mutation operators using TXL, but we expect it to be useful for implementing complex mutation operators in the future.

#### *6.3.2. The test runner*

The test runner of SMT-C is built as a front end for a C unit testing tool: Check [50]. Check is based on Autotools [56]. We used the Autotools plug-in for CDT to connect SMT-C with Check.

In SMT-C, tests are implemented according to the requirements of Check and these are relatively easy to follow. For example, a test case is written between a pair of predefined C macros as follows:

```
START_TEST (test_name)
{
  /* unit test code */
}
END_TEST
```

Our test runner inherits advanced features from Check such as run in fork mode, test fixture, multiple suites in one runner, looping tests, test time-outs, determining test coverage, and XML logging.

## 7. An experimental study

To investigate the basic features of semantic mutation testing, we applied the 13 semantic mutation operators to a set of 8 subject programs. To learn the differences between semantic mutation testing and traditional mutation testing, we also applied 7 selected traditional mutation operators to the subject programs. In the experiment, we compared the number of mutants generated by semantic/tradition mutation operators, mutation scores of operators, and average semantic sizes of operators. We also investigated subsumes relations between semantic/traditional mutation operators.

### *7.1. Experiment settings*

The subject programs were chosen from Software-artifact Infrastructure Repository (SIR) as shown in Table 1 [57]. We chose these programs since they are well developed test objects for which there are test suites. For each program $p$ we created a universal test suite, which is the set of test cases that SIR has for $p$.

The selected traditional operators were as follows:

1. **OAAN**: arithmetic operator to another arithmetic operator;
2. **SCRB**: *continue* to *break*;
3. **SBRC**: *break* to *continue*;

**Table 1**
Subject programs.

| Program | Printtokens | Printtokens2 | Replace | Schedule | Schedule2 | Space | Tcas | Totinfo |
|---|---|---|---|---|---|---|---|---|
| NLOC | 343 | 355 | 513 | 296 | 263 | 5905 | 137 | 281 |
| NUTS | 4130 | 4115 | 5542 | 2650 | 2710 | 13585 | 1608 | 1052 |

NLOC and NUTS are the abbreviations of number of lines of executable codes and number of test cases in universal test suite.

**Table 2**
Mutants generated by semantic mutation operators.

| **Program** | **AOR** | **IMB** | **LBC_C** | **LBC_I** | **LBM_C** | **LBM_I** | **MFC_E** | **MFC_R** | **Total** | **Killed** | **Live** | **Score** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| print_tokens | | 12 | | 2 | | 2 | | | 16 | 16 | | 1.0000 |
| print_tokens2 | | | | 1 | | 1 | | | 2 | 2 | | 1.0000 |
| replace | | 3 | | | | | | | 3 | 3 | | 1.0000 |
| schedule | | | 1 | | 1 | | | | 2 | 2 | | 1.0000 |
| schedule2 | 2 | | 1 | | 1 | | | | 4 | 3 | 1 | 0.7500 |
| space | 3 | 1 | 6 | 5 | 6 | 5 | | 4 | 30 | 14 | 16 | 0.4667 |
| tcas | | | | | | | | | 0 | 0 | | 0.0000 |
| tot_info | | | | | | | 1 | 10 | 11 | 2 | 9 | 0.1818 |
| Total | 5 | 16 | 8 | 8 | 8 | 8 | 1 | 14 | 68 | 42 | 26 | 0.6176 |
| Killed | 5 | 15 | 5 | 6 | 2 | 6 | | 3 | 42 | | | |
| Live | | 1 | 3 | 2 | 6 | 2 | 1 | 11 | 26 | | | |
| Score | 1.0000 | 0.9375 | 0.6250 | 0.7500 | 0.2500 | 0.7500 | 0.0000 | 0.2143 | 0.6176 | | | |

4. **STRP**: trap on statements;
5. **STRI**: trap on *if* conditions;
6. **SSWM**: trap on *switch* cases;
7. **SSDL**: statement deletion.

We chose these operators because they were syntactically related to the semantic operators proposed in this paper. For example, **LBM_I** modifies *if* statements that contain no *else* part and these may also be changed by **STRI** operators.

All experiments were conducted with SMT-C. To use these subject programs in SMT-C, an Eclipse C project using Autotools plug-in was created for each subject program. For running the test suites, test drivers were implemented according to the requirements of Check. When applying a semantic mutation operator *op* we can make all changes to the semantics of affected constructs to produce one mutant. However, this would have given us only a small number of semantic mutants and so instead we produced one mutant for each construct in a program whose semantics is affected by *op*. We now describe the results of the experiments under two categories: experiments that explored the nature of the semantic mutation operators and experiments that compared operators.

### 7.2. Properties of semantic mutation operators

In calculating the mutation scores of the proposed operators, the number of killed mutants was the sum of the number of mutants that could not be compiled and the number of mutants that were killed by running the universal testing suite. The *mutation score* given in this paper is the number of killed mutants divided by the total number of mutants. This is actually a lower bound on the real mutation score since the number of equivalent mutants should be subtracted from the total number of mutants. Section 8 provides a brief discussion regarding equivalent mutants generated by semantic mutation operators. For traditional mutation operators, the approximation is necessary because of the large number of live mutants; it simply was not feasible to manually investigate these (2293) mutants and automatically detecting equivalent mutants is an undecidable problem [3,58]. The results for mutants generated by semantic/traditional mutation operators are given in Tables 2 and 3. We note that there are only 8 semantic operators in Table 2. This is because 5 operators (**ASD**, **DIA_F**, **DIA_T**, **FTA_F** and **FTA_T**) did not generate any mutants when applied to the 8 subject programs.

The data presented in Tables 2 and 3, shows that the 13 semantic mutation operators generated 68 mutants for the 8 subject programs. That is far fewer than the 12598 mutants generated by 7 traditional mutation operators. In addition, 5 semantic mutation operators generated no mutants. This is because semantic mutation mutates certain features associated with specific types of faults and these may not be present.

We also note that average mutation score of semantic mutation operators (0.6176) is lower than that of the traditional mutation operators (0.8180). This is because some of the semantic operators generated mutants that were really hard to kill. For example, **LBM_C**, **MFC_E** and **MFC_R** have mutation score less than or equal to 0.25. When compared with syntactic mutation testing, the tester has to deal with far fewer mutants but some of these mutants may be more difficult to kill than typical syntactic mutants.

To calculate the average semantic size of a mutation operator, we recorded the number of test cases that killed the mutants generated by an operator. Let us suppose that operator *op* generated the set $\{m_i : i \in [1, n]\}$ of mutants. Further,

**Table 3**
Mutants generated by traditional mutation operators.

| Program | OAAN | SCRB | SBRC | STRP | STRI | SSWM | SSDL | Total | Killed | Live | Score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| print_tokens | 32 | | 9 | 224 | 23 | 54 | 224 | 566 | 528 | 38 | 0.9329 |
| print_tokens2 | 36 | | | 231 | 63 | | 231 | 561 | 466 | 95 | 0.8307 |
| replace | 152 | | 10 | 258 | 44 | 6 | 258 | 728 | 728 | 0 | 1.0000 |
| schedule | 52 | | 7 | 145 | 18 | 7 | 145 | 374 | 345 | 29 | 0.9225 |
| schedule2 | 36 | | 11 | 139 | 22 | 10 | 139 | 357 | 301 | 56 | 0.8431 |
| space | 1088 | 1 | 40 | 3890 | 466 | 16 | 3890 | 9391 | 7359 | 2032 | 0.7836 |
| tcas | 4 | | | 61 | 7 | | 61 | 133 | 122 | 11 | 0.9173 |
| tot_info | 228 | 2 | | 119 | 20 | | 119 | 488 | 456 | 32 | 0.9344 |
| Total | 1628 | 3 | 77 | 5067 | 663 | 93 | 5067 | 12598 | 10305 | 2293 | 0.8180 |
| Killed | 1439 | 3 | 70 | 4500 | 560 | 93 | 3640 | 10305 | | | |
| Live | 189 | 0 | 7 | 567 | 103 | 0 | 1427 | 2293 | | | |
| Score | 0.8839 | 1.0000 | 0.9091 | 0.8881 | 0.8446 | 1.0000 | 0.7184 | 0.8180 | | | |

**Table 4**
Semantic size of mutation operators.

| Semantic mutation operators | | | | | Selected traditional mutation operators | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Operator** | **KTC** | **TTC** | **SS** | **SS*** | **Operator** | **KTC** | **TTC** | **SS** | **SS*** |
| AOR | 21190 | 40755 | 0.5199 | 0.5199 | OAAN | 1009749 | 12422693 | 0.0813 | 0.2809 |
| IMB | 15641 | 79777 | 0.1961 | 0.1274 | SBRC | 19195 | 109835 | 0.1748 | 0.4122 |
| LBC_C | 3131 | 86870 | 0.0360 | 0.2395 | SCRB | 1115 | 15689 | 0.0711 | 0.2529 |
| LBC_I | 7241 | 80300 | 0.0902 | 0.3032 | SSDL | 7234602 | 55431596 | 0.1305 | 0.2889 |
| LBM_C | 129 | 86870 | 0.0015 | 0.0033 | SSWM | 332771 | 508694 | 0.6542 | 0.6518 |
| LBM_I | 2981 | 80300 | 0.0371 | 0.1399 | STRI | 1112628 | 7063097 | 0.1575 | 0.3151 |
| MFC_R | 1337 | 64860 | 0.0206 | 0.0369 | STRP | 17272962 | 57116847 | 0.3024 | 0.5092 |
| **Total** | **51650** | **519732** | **0.0994** | **0.1957** | **Total** | **26983022** | **1.33E+08** | **0.2034** | **0.3873** |

KTC is the number of killed test cases; TTC is the total number of test cases run;
SS is the semantic size; SS* is the mean of the averages of the semantic size.

for mutant $m_i$, let us suppose that $k_i$ test cases out of the total $t_i$ test cases in the universal test suite kill $m_i$. The average semantic size of $op$, $ss_{op}$, is calculated as follows:

$$ss_{op} = \frac{\Sigma_{i=1}^{n} k_i}{\Sigma_{i=1}^{n} t_i} \tag{1}$$

An alternative approach is to calculate the mutation scores for each program and take the average of these scores. This avoids biasing the results in favor of a small number of larger programs. We computed both values, calling the former SS and the latter SS*, which is defined as follows:

$$ss^*_{op} = \frac{1}{n} \Sigma_{i=1}^{n} \frac{k_i}{t_i} \tag{2}$$

We also calculated the average semantic size of the two sets of operators: semantic mutation operators and selected traditional mutation operators. This was computed in a similar manner to that of the individual operators, simply computing over all mutants formed by a set of operators rather than all mutants produced by a particular operator. These figures are in the last row of Table 4.

The hypothesis, that semantic mutants are harder to kill than syntactic mutants, is partially supported by the semantic size data given in Table 4. Consider the value of SS; a similar pattern is found with SS*. The average semantic size of mutants produced by semantic mutation operators was 0.0994 which is only a half of that found with traditional syntactic mutation operators (0.2034). We can also see that **LBM_C** and **MFC_R** both had very low average semantic sizes, 0.0206 and 0.0015 respectively, while the smallest average semantic size in the traditional mutation operator set was 0.0711 (**SCRB**).

### 7.3. Comparing operators

For all 20 operators used in the experiments, a $20 \times 20$ subsume matrix was produced to investigate the subsume relations between any two of the operators. Let us say that killable mutants are mutants that can be built and can be distinguished from the original program by running the corresponding universal test suite. For a killable mutant $m$, the killing test suite of $m$ is the subset of the universal test suite that contains the test cases that kill $m$.

For a set of operators, $OP = \{op_i : i \in [1, l]\}$, the subsume table is an $l \times l$ matrix $S$. To fill the subsume matrix, we did the following for each pair of operators $(op_i, op_j)$, where $i, j \in [1, p]$:

- Generate a minimised selective test suite $T_i$ for $op_i$;
- Run the test cases from $T_i$ on each killable mutant generated by $op_j$;

**Table 5**
Subsume table.

| Operator | AOR | IMB | LBC_C | LBC_I | LBM_C | LBM_I | MFC_R | OAAN | SCRB | SBRC | STRP | STRI | SSWM | SSDL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **AOR** | 1 | 0 | 0.1667 | 0.0667 | 0.05 | 0.05 | 0.4 | 0.2892 | 0 | 0.5833 | 0.5211 | 0.2569 | 1 | 0.3715 |
| **IMB** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0.7798 | 0 | 0.8 | 0.8635 | 0.6415 | 1 | 0.8471 |
| **LBC_C** | 0.625 | 0 | 1 | 0.1167 | 0.6 | 0.1 | 0.8 | 0.2401 | 0 | 0.2833 | 0.5283 | 0.2641 | 0.9969 | 0.4272 |
| **LBC_I** | 0 | 0.5958 | 0.2 | 1 | 0.05 | 0.8833 | 0 | 0.2259 | 0 | 0.6214 | 0.5242 | 0.2681 | 0.9543 | 0.3174 |
| **LBM_C** | 0 | 0 | 0.3625 | 0.0833 | 1 | 0.0833 | 0.05 | 0.154 | 0 | 0.175 | 0.4042 | 0.1585 | 0.9531 | 0.233 |
| **LBM_I** | 0 | 0.2458 | 0.3 | 1 | 0.1 | 1 | 0.1 | 0.2374 | 0 | 0.55 | 0.5043 | 0.2534 | 0.8521 | 0.3147 |
| **MFC_R** | 0 | 0 | 0.3333 | 0.0167 | 0 | 0.0167 | 1 | 0.2931 | 0 | 0.3167 | 0.4328 | 0.1954 | 0.9594 | 0.3697 |
| **OAAN** | 1 | 0.9367 | 0.725 | 0.9833 | 0.375 | 0.9833 | 1 | 1 | 1 | 0.625 | 0.877 | 0.6943 | 1 | 0.8258 |
| **SCRB** | 0 | 0 | 0.05 | 0 | 0 | 0 | 0.3333 | 0.2303 | 1 | 0.7667 | 0.4348 | 0.1968 | 0.925 | 0.2713 |
| **SBRC** | 0.575 | 0.775 | 0.0875 | 0.15 | 0.025 | 0.1 | 0.2 | 0.2086 | 0.1 | 1 | 0.5536 | 0.3101 | 0.9764 | 0.3734 |
| **STRP** | 1 | 1 | 0.7875 | 1 | 0.525 | 1 | 1 | 0.9912 | 1 | 0.705 | 1 | 0.9998 | 1 | 0.9875 |
| **STRI** | 1 | 0.99 | 0.7125 | 0.8833 | 0.55 | 0.8833 | 0.9333 | 0.9514 | 1 | 0.62 | 0.9949 | 1 | 1 | 0.9709 |
| **SSWM** | 0.925 | 0.8967 | 0.3 | 0.4833 | 0.075 | 0.4333 | 0.85 | 0.5034 | 0.05 | 0.53 | 0.6901 | 0.4325 | 1 | 0.5721 |
| **SSDL** | 1 | 1 | 0.8 | 1 | 0.525 | 1 | 0.9833 | 0.9996 | 1 | 0.675 | 0.9966 | 0.9952 | 1 | 1 |
| **TSS** | 4.9 | 14.6 | 5 | 6 | 2 | 5 | 2.95 | 485.15 | 2.8 | 10.9 | 1742 | 484.1 | 66.75 | 1719 |

TSS means the number of total selective test cases.

- count the numbers of killed mutants and total killable mutants;
- calculate $S[i][j]$ as the ratio of these two numbers.

Let us suppose that the set of killable mutants for operator $op_k$ is denoted by $M_k$. In the first step $T_i$ is generated by repeating the following until $M_i$ is empty: randomly pick one test case $t$ that kills a mutant in $M_i$, add $t$ to $T_i$, and then remove from $M_i$ all mutants killed by $t$. To calculate $S[i][j]$, let us suppose that $a$ is the number of mutants of $op_j$ killed by $T_i$ and $b$ is the total number of killable mutants of $op_j$, then $S[i][j] = a/b$. After the procedure, every item in $S$ is filled. If $S[i][j] = 1$, then $op_i$ subsumes $op_j$: the test suite $T_i$ produced to kill the mutants generated by $op_i$ also kills the mutants produced by $op_j$. According to [59,60], $S[i][j]$ close to 1 suggests that $op_i$ ProbSubsumes $op_j$. We note that this algorithm includes some randomisation and therefore we can obtain different values if we repeat the process. Therefore, we ran this process 20 times and averaged the values.

For each pair of operators $op_i$ and $op_j$ we produced a cumulative value for $S[i][j]$, across all 8 programs, in the following way. For a program $p$ we produced a test suite $T_i^p$ that kills the mutants of $p$ produced using $op_i$ as before. Let us suppose that $a^p$ is the number of mutants of $p$ produced using $op_j$ that are killed by $T_i^p$ and $b^p$ is the total number of killable mutants of $op_j$. Again, these are averaged over 20 randomly chosen $T_i^p$. Let $a$ be the sum of the $a^p$ over the 8 programs and let $b$ be the sum of the $b^p$ over the 8 programs. Then we let $S[i][j] = a/b$.

The results of the overall subsume matrix is given in Table 5. We note that this is a $14 \times 14$ matrix since 5 semantic mutation operators produce no mutants and one semantic mutation operator generated only live mutants. The last row of the table gives the average size of the selective test suites for the corresponding operator. For example, the first cell in the last row is 4.9 which means that the average size of a selected test suite which kills all **AOR** mutants is 4.9.

Regarding the relationships between operators, we first consider subsume relations between two semantic mutation operators. According to the overall subsume matrix given in Table 5, there is one such subsume relation: **LBM_I** is subsumed by **LBC_I**. This is reasonable since **LBM_I** changes the last *else if* statement to *else* and **LBC_I** adds missing *else* statements; if both operators modify the same *if ... else if* statement, a test case that kills the mutant generated by **LBC_I** will execute the statements changed by **LBM_I**. Apart from this, no other pairs of semantic mutation operators were related under subsumes. This is ideal since the semantic mutation operators are designed to simulate different possible faults in C programs.

In the set of traditional mutation operators, there are more examples where the subsume relation holds. For example, two of the traditional mutation operators were subsumed by **STRP**. In addition, if we assume that a mutation score higher than 0.95 suggests that two operators have the ProbSubsume relation we find that **STRP** ProbSubsumes **OOAN**, **STRI** and **SSDL**. However, the subsume relation involving **STRP** is not very helpful since **STRP** generates so many mutants and the size of the selected test suite of **STRP** is the largest (1742). Naturally, there is a relationship between test suite size and effectiveness [61] and this relationship may explain the results observed.

When comparing the sets of semantic mutation operators and traditional mutation operators, we find that 6 operators (**AOR**, **IMB**, **LBC_C**, **LBC_I**, **LBM_C** and **MFC_R**) subsume or ProbSubsume operator **SSWM**. This is understandable since the mutants generated by **SSWM** had a large mean semantic size (0.6642) as shown in Table 4. Other than this, there were no subsume relations from semantic mutation operators to traditional mutation operators. From traditional mutation operators to semantic mutation operators, except **LBC_C** and **LBM_C**, the other 5 semantic operators are ProbSubsumed by **OAAN**, **STRP**, **STRI** and **SSDL**. Again, this may be because these 4 traditional operators generate so many mutants and lead to large selected test suite (more than 480 test cases).

It is also interesting to investigate the overall subsume relation between the set of semantic mutation operators ($OP_S$) and the set of traditional mutation operators ($OP_T$). The algorithm is similar to calculating the subsume relation between two

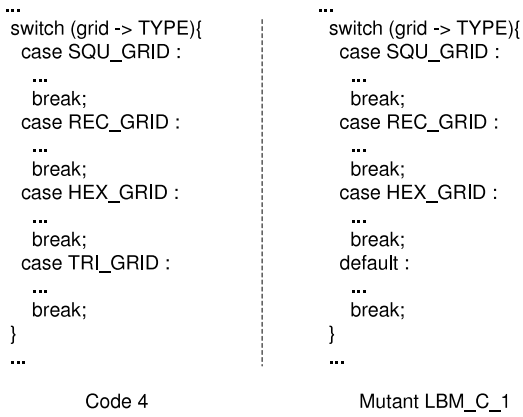


**Fig. 9.** A piece of code from *space* and its mutant.

operators except that we look at all mutants produced by one of these sets of operators, rather than all mutants produced by a single operator. Again, we averaged scores over 20 runs of the algorithm. The final average mutation score which represents the subsume relation from $OP_S$ to $OP_T$ is 0.5978 and the size of the selected test suite is 40. Using the same approach to calculate whether $OP_T$ subsumes $OP_S$, the final average mutation score is 0.9585 and the size of the selected test suite is 3021.8.

It is clear that there were no subsume relationships between the sets of syntactic mutation operators and the semantic mutation operators. However, syntactic mutation operators do ProbSubsume semantic mutation operators. This may be because of the large number of mutants generated by traditional mutation operators and the corresponding large test suites used. In addition, a relatively large proportion of the mutants generated by semantic mutation operators were live and these mutants were ignored when analyzing subsume relations.

## 8. Discussion

We manually analyzed the 26 live semantic mutants shown in Table 2. We found that except for the mutants generated by **MFC_E** and **MFC_R** all other live mutants were equivalent. For example, we compare a piece of code from the original *space* program with its mutant *LBM_C_1* in Fig. 9.

The difference is that the last *case* statement in the original program has been change to a *default* branch. According to a manual review of the program, the whole program ensured that variable *grid*→*TYPE* cannot be assigned to a value other than *SQU_GRID*, *REC_GRID*, *HEX_GRID* or *TRI_GRID*, so the two pieces of code are equivalent. However, if we were to only consider the scope of the function that contains the piece of code, we find that the two functions are not equivalent. One of the integer parameters of the function is directly assigned to *grid*→*TYPE*. It can be argued that this could correspond to a fault in future if this piece of code is changed since equivalence depends on the context in which the code lies.

All 11 live mutants generated by **MFC_E** and **MFC_R** for subject program *tot_info* are equivalent. However, it may be worth noting some interesting observations. We found that manually generated test cases caused 3 mutants to generate different intermediate results from the original program, but this difference cannot be revealed because the float numbers are truncated in the output procedure. Another observation is that 3 other mutants are killed by the universal test suite when using different optimisation options at compilation. For the 4 mutants generated by **MFC_R** of subject program *space*, we found that 3 of them are not equivalent since they can be killed by manually generated test cases. This observation implies that the universal test suite of *space* program contains potential limitations and the **MFC_R** operator cannot be subsumed by the other operators used in this experiment.

## 9. Threats to validity

There are several threats to the validity of the experiments. Internal threats relate to the possibility that other factors have influenced the results and that any differences observed are a consequence of such factors. We have tried to avoid this possibility by using the same programs and test suites throughout the experiments.

Threats of construct validity refer to the potential for mistakes in the measurement. We reduced the scope for such mistakes by building SMT-C on top of widely used tools such as Eclipse and Check. We also tested SMT-C and performed a few initial small experiments, that created and tested mutants, and manually checked the results.

There are clear difficulties in generalising the results of the experiments and these form threats to external validity. We have only used a few, relatively small programs and these have all been taken from the same source. We have tried to reduce this threat by using experimental subjects that have been used in many previous experiments and are generally seen as valuable subjects. However, it is clear that there is a need to extend the experiments to additional programs and ideally programs from different sources.

## 10. Conclusions

This paper has introduced semantic mutation testing (SMT), which is a fundamentally new type of mutation testing where we mutate the semantics of the language used rather than the syntax of the description. The aim is to represent potential misunderstandings of the semantics of a description language. We have described a range of scenarios in which semantic mutation testing may play a significant role and also have described a semantic mutation testing tool that has been developed.

Traditional mutation testing mutates the syntax of a description $N$ to form some mutant $N'$. The mutant is usually produced by the application of a mutation operator. $N'$ is killed by a test case if the test case distinguishes between $N$ and $N'$. A test suite is sufficient if it kills every (non-equivalent) mutant formed. The idea is that the mutants simulate possible mistakes and thus that a test suite that kills the non-equivalent mutants will be good at finding such mistakes. However, the behavior associated with a description is defined by a combination of the syntax $N$ of the description and the semantics $L$ of the language in which it is described. Thus, traditional mutation operators provide a mapping of the form $(N, L) \rightarrow (N', L)$.

In SMT the semantics of the description language $L$ are mutated. Thus, the application of a semantic mutation operator is of the form $(N, L) \rightarrow (N, L')$. Semantic mutation testing aims to simulate mistakes that are a consequence of a misunderstanding of the semantics of the language used.

It is argued that SMT captures a different type of mistake to traditional mutation testing. The error model of semantic mutation and a number of scenarios, in which SMT is of particular value, have been outlined. In one scenario, a company has migrated from one tool to another, and thus from one semantics to another. Here errors might result from the use of aspects of the previous semantics with the new tool. SMT could be used to investigate such issues and so the process of migrating from one toolset to another could be supported by a set of semantic mutation operators.

Examples in a high-level specification language, statecharts, and a low-level programming language, C, were given to show the capability of SMT. Interestingly, in the statechart example, we found that some standard state-based test criteria might fail to find the differences. By investigating different ways of implementing undefined or unspecified elements of a language (such as the comparison of floating-point numbers), SMT may be used to explore the portability of code written in C. Semantic mutation operators can also be designed to target misunderstandings caused by the use of the same syntactic construct in the specification and code. Ideally, the semantic mutation operators used depends upon the development process that has been applied and the background of the developers, since these will influence the likely mistakes.

One potential benefit of SMT is that for each mutation operator we obtain one mutant since we are mutating the semantics of the description language, not parts of the description. However, at times there may be value in mutating the semantics of only parts of a description. For example, a piece of software may have been developed by several people with only one of them having a background that indicates that a particular semantic mutation operator should be applied. One of the advantages of the proposed approach to implementing semantic mutation operators, which is by simulating them through making syntactic changes, is that it is relatively straightforward to mutate the semantics of only some parts of a description.

We described a semantic mutation testing tool, SMT-C, for C code. This uses TXL in order to implement the mutation operators and is designed to ensure that it is relatively straightforward to implement addition operators. Interestingly, it transpired that some of the operators were relatively difficult to implement since they required type information and as a result we devised a type annotation module for TXL. We ran experiments to investigate the nature of a set of semantic mutation operators and to compare these with traditional mutation operators. We found that semantic mutation operators produced far fewer mutants than the syntactic mutation operators; 68 in total as opposed to 12598. This suggests that it may be easier to scale such an approach to semantic mutation and it might be applied to larger programs. In addition, the average semantic size of the semantic mutants was half that of the syntactic mutants.

We compared 7 semantic operators and 7 syntactic operators using the subsume relation. Within the set of semantic mutation operators we found that one operator was subsumed by another. This suggests that our attempt to produce operators that represent very different types of faults was reasonably successful. We found many more instances of the subsume relations within the set of syntactic mutation operators. However, we found that test suites that kill all of the syntactic mutants also killed approximately 95% of the semantic mutants. In contrast, test suites that kill all of the semantic mutants killed approximately 59% of the syntactic mutants. However, this may largely be due to the differences in size of these test suites: the reduced test suites produced to kill the semantic mutants contained 40 test cases on average while those produced to kill syntactic mutants contained 3021.8 test cases on average. The fact that neither set subsumed the other suggests that ideally both semantic and syntactic mutants should be used.

The experiments concentrated on the effectiveness of testing and not on efficiency. However, most of the semantic mutation operators are no more complex to apply than the traditional mutation operators and they are applied far fewer times. The exceptions are those operators that require us to have information about the types of variables: for such operators the current version of the SMT-C tool is less efficient and increases the number of nodes in the parse tree. However, we expect to be able to make this much more efficient: currently it is not a significant issue in running the experiments since the test execution time dominates the time taken to generate the semantic mutants.

There are many avenues for future work. In particular, there is the need to develop additional semantic mutation operators and perform more experiments. It would be particularly interesting to investigate the effectiveness of semantic mutation testing in situations where there is reason to believe that particular semantic misunderstandings are likely, for

example, when developers are migrating between languages or tools. The hope is that in such situations semantic mutants will correspond exactly to likely faults and thus that semantic mutation testing will help the tester to find such faults.

## References


[1] A.J. Offutt, J.H. Hayes, A semantic model of program faults, in: International Symposium on Software Testing and Analysis, ISSTA 1996 1996, pp. 195–200.
[2] J.A. Clark, H. Dan, R.M. Hierons, Semantic mutation testing, in: Fourth Workshop on Mutation Analysis, 2010, pp. 100–109.
[3] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on test data selection: help for the practical programmer, IEEE Computer 11 (4) (1978) 31–41.
[4] R.G. Hamlet, Testing programs with the aid of a compiler, IEEE Transactions on Software Engineering 3 (1977) 279–290.
[5] W.E. Howden, Weak mutation testing and completeness of test sets, IEEE Transactions on Software Engineering 8 (4) (1982) 371–379.
[6] M.R. Woodward, K. Halewood, From weak to strong, dead or alive? an analysis of some mutation testing issues, in: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, Banff, Canada, 1988.
[7] A.J. Offutt, Investigations of the software testing coupling effect, ACM Transactions on Software Engineering Methodology 1 (1) (1992) 3–18.
[8] K.S.H.T. Wah, A theoretical study of fault coupling, Journal of Software Testing, Verification and Reliability 10 (1) (2000) 3–45.
[9] J.H. Andrews, L.C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: 27th International Conference on Software Engineering, ICSE 2005, 2005, pp. 402–411.
[10] D. Baldwin, F. Sayward, Heuristics for determining equivalence of program mutations, Research Report 276, Department of Computer Science, Yale University.
[11] B. Baudry, F. Fleurey, J.-M. Jezequel, Y.L. Traon, From genetic to bacteriological algorithms for mutation-based testing, Journal of Software Testing, Verification and Reliability 15 (2) (2005) 73–96.
[12] L. Bottaci, E.S. Mresa, Efficiency of mutation operators and selective mutation strategies: an empirical study, Journal of Software Testing, Verification and Reliability 9 (4) (1999) 205–232.
[13] T.A. Budd, Mutation analysis: ideas, examples, problems and prospects, in: Proceedings of the Summer School on Computer Program Testing, Sogesta, 1981, pp. 129–148.
[14] P.G. Frankl, S.N. Weiss, C. Hu, All-uses vs mutation testing: an experimental comparison of effectiveness, Journal of Systems Software 38 (3) (1997) 235–253.
[15] M. Harman, R.M. Hierons, S. Danicic, The relationship between program dependence and mutation analysis, in: First Workshop on Mutation Analysis, Mutation 2000, Kluwer, San Jose, California, USA, 2001, pp. 5–13.
[16] R.M. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Journal of Software Testing, Verification and Reliability 9 (4) (1999) 233–262.
[17] S. Kim, J. Clark, J. McDermid, Assessing test set adequacy for object-oriented programs using class mutation, in: Symposium on Software Technology, SoST 1999, 1999, pp. 72–83.
[18] K. Kapoor, J.P. Bowen, Ordering mutants to minimise test effort in mutation testing, in: Formal Approaches to Software Testing, FATES 2004, 2004, pp. 195–209.
[19] K.N. King, A.J. Offutt, A Fortran language system for mutation-based software testing, Software—Practice and Experience 21 (7) (1991) 685–718.
[20] Y.-S. Ma, M.J. Harrold, Y.R. Kwon, Evaluation of mutation testing for object-oriented programs, in: 28th International Conference on Software Engineering, ICSE 2006, 2006, pp. 869–872.
[21] A.J. Offutt, W.M. Craft, Using compiler optimization techniques to detect equivalent mutants, Journal of Software Testing, Verification and Reliability 4 (3) (1994) 131–154.
[22] A.J. Offutt, S.D. Lee, An empirical evaluation of weak mutation, IEEE Transactions on Software Engineering 20 (5) (1994) 337–344.
[23] A.J. Offutt, J. Pan, Detecting equivalent mutants and the feasible path problem, in: Annual Conference on Computer Assurance, COMPASS 1996, IEEE Computer Society Press, Gaithersburg, MD, 1996, pp. 224–236.
[24] A.J. Offutt, J. Pan, K. Tewary, T. Zhang, An experimental evalutation of data flow and mutation testing, Software—Practice and Experience 26 (2) (1996) 165–176.
[25] J. Offutt, J. Pan, Automatically detecting equivalent mutants and infeasible paths, Software Testing, Verification, and Reliability 7 (3) (1997) 165–192.
[26] J. Voas, G. McGraw, Software Fault Injection, Wiley, 1998.
[27] P.E. Black, V. Okun, Y. Yesha, Mutation of model checker specifications for test generation and evaluation, in: First Workshop on Mutation Analysis, Mutation 2000, Kluwer Academic Publishers, San Jose, California, 2000, pp. 14–20.
[28] S. Fabbri, J. Maldonado, T. Sugeta, P. Masiero, Mutation testing applied to validate specifications based on statecharts, in: 10th International Symposium on Software Reliability Engineering, ISSRE 1999, IEEE Press, 1999, pp. 210–219.
[29] T. Sugeta, J.C. Maldonado, W.E. Wong, Mutation testing applied to validate SDL specifications, in: 16th IFIP International Conference on Testing of Communicating Systems, TestCom 2004, 2004, pp. 193–208.
[30] M.R. Woodward, Errors in algebraic specifications and an experimental mutation testing tool, IEE/BCS Software Engineering Journal 8 (4) (1993) 211–224.
[31] Y. Zhan, J.A. Clark, Search-based mutation testing for *simulink* models, in: Genetic and Evolutionary Computation Conference, GECCO 2005, 2005, pp. 1061–1068.
[32] J. Srivratanakul, J. Clark, F. Polack, S. Stepney, Challenging formal specifications with mutation: a CSP security example, in: 12th IEEE Asia Pacific Software Engineering Conference, APSEC 2003, 2003.
[33] A.S. Namin, J.H. Andrews, Finding sufficient mutation operators via variable reduction, in: Second Workshop on Mutation Analysis, Mutation 2006, 2006.
[34] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, C. Zapf, An experimental determination of sufficient mutant operators, ACM Transactions on Software Engineering and Methodology 5 (2) (1996) 99–118.
[35] A.J. Offutt, A. Lee, G. Rothermel, C. Zapf, An experimental evaluation of selective sufficient mutant, in: 15th International Conference on Software Engineering, ICSE 1993, IEEE Press, Baltimore, 1993, pp. 100–117.
[36] R. Alur, M. Yannakakis, Model checking of message sequence charts, in: Proceedings of 10th International Conference on Concurrency Theory, Eindhoven, Netherlands, 1999, pp. 114–29.
[37] S.S. Brilliant, J.C. Knight, N.G. Leveson, Analysis of faults in an n-version software experiment, IEEE Transactions on Software Engineering 16 (2) (1990) 238–247.
[38] ISO, ISO/IEC 13568:2002: Information technology – Z formal specification notation – syntax, type system and semantics, ISO, 2002.
[39] J.G.P. Barnes, Ada 95 Rationale, The Language, The Standard Libraries, Springer–Verlag, 1997.
[40] P. McMinn, Search-based failure discovery using testability transformations to generate pseudo-oracles, in: Genetic and Evolutionary Computation Conference, GECCO 2009, 2009, pp. 1689–1696.
[41] M. von der Beeck, A comparison of statechart variants, in: Formal Techniques in Real–Time and Fault–Tolerant Systems, in: LNCS, vol. 863, Springer–Verlag, 1994, pp. 128–148.
[42] D. Harel, M. Politi, Modeling Reactive Systems with Statecharts: The STATEMATE Approach, McGraw-Hill, New York, 1998.
[43] S. Fabbri, J. Maldonado, T. Sugeta, P. Masiero, Requirements-level semantics for UML statecharts, in: Formal Methods for Open Object-Based Distributed Systems, FMOODS 2000, Kluwer Academic Press, Stanford, California, USA, 2000, pp. 121–140.
[44] D.P. Sidhu, T.-K. Leung, Formal methods for protocol testing: a detailed study, IEEE Transactions on Software Engineering 15 (4) (1989) 413–426.

[45] D. Harel, A. Naamad, The STATEMATE semantics of statechart, ACM Transactions on Software Engineering and Methodology 5 (4) (1996) 293–333.
[46] MathWorks Inc., Stateflow User's Guide, 4th edition, 2001.
[47] L. Hatton, Safer C: Developing Software in High-integrity and Safety-critical Systems, McGraw–Hill, 1994.
[48] A. X3J9, ISO/IEC 9899-1999, Programming Languages—C, ISO, 1999.
[49] J. Cordy, TXL-a language for programming language tools and applications, Electronic Notes in Theoretical Computer Science 110 (2004) 3–31.
[50] Check: a unit testing framework for C, http://check.sourceforge.net/, July 2010.
[51] I. 13568, Information technology – Z formal specification notation – syntax, type system and semantics, 2002.
[52] A. X3J11/88-090, Draft ANSI C Standard, 5 1988.
[53] B. Stroustrup, The C++ Programming Language, Addison-Wesley, 1991.
[54] J. Gosling, B. Joy, G. Steele, G. Bracha, *Java*$^{TM}$ Language Specification, Addison-Wesley, 2005.
[55] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. Martin, A. Mathur, E. Spafford, Design of mutant operators for the C programming language, Tech. Rep., Department of Computer Sciences, Purdue University, 1989.
[56] Autoconf, http://www.gnu.org/software/autoconf/, July 2010.
[57] H. Do, S.G. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact, Empirical Software Engineering: An International Journal 10 (4) (2005) 405–435.
[58] P. Frankl, O. Iakounenko, Further empirical studies of test effectiveness, in: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Orlando USA, 1998, pp. 153–162.
[59] A. Mathur, W. Wong, An empirical comparison of data flow and mutation-based test adequacy criteria, Software Testing, Verification and Reliability 4 (1) (1994) 9–31.
[60] W. Wong, On mutation and data flow, Ph.D. thesis, Purdue University, 1993.
[61] A.S. Namin, J.H. Andrews, The influence of size and coverage on test suite effectiveness, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, 2009, pp. 57–68.