2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

SMT-C: A Semantic Mutation Testing Tool for C

Haitao Dan and Robert M. Hierons
School of Information Systems, Computing & Mathematics
Brunel University
Uxbridge, Middlesex UB8 3PH, UK
{haitao.dan,rob.hierons}@brunel.ac.uk

***Abstract*—Semantic Mutation Testing (SMT) is a technique that aims to capture errors caused by possible *misunderstandings* of the semantics of a description language. It is intended to target a class of errors which is different from those captured by traditional Mutation Testing (MT). This paper describes our experiences in the development of an SMT tool for the C programming language: SMT-C. In addition to implementing the essential requirements of SMT (generating semantic mutants and running SMT analysis) we also aimed to achieve the following goals: weak MT/SMT for C, good portability between different configurations, seamless integration into test routines of programming with C and an easy to use front-end.**

***Keywords*-Semantic mutation testing; Mutation operator; Unit test; Weak mutation testing; Eclipse plugin**

I. Introduction

Mutation Testing (MT) is a powerful and flexible testing technique [1], [2], [3], [4], [5], [6]. In traditional MT, simple faults are injected into a program by making syntactical changes with mutation operators. The resulting programs are called *mutants*. Semantic Mutation Testing (SMT) was recently proposed to tackle a specific type of mistakes [7], [8]. The hypothesis behind SMT is that the faults introduced by possible *misunderstandings* of the language used to deliver software artifacts can be represented by mutating the semantics of the language. In this paper, we introduce an SMT tool under development for assessing the SMT hypothesis, **SMT-C**. It is designed to be integrated into the software engineering development process, easy-to-use, flexible and with good portability.

There are different ways to implement semantic misunderstandings into a program [8]. The most intuitive way is to simulate a semantic mutation by making changes to the syntax of the description. The syntactical approach was adopted in the development of **SMT-C**. Therefore, like general MT tools, **SMT-C** introduces faults into a C program, but mutants generated by **SMT-C** simulate misunderstandings of the C semantics. Consequently, the new tool has similar functionality to other MT tools for C [9], [10], [11], [12], [13], but we had to implement **SMT-C** from scratch since special requirements were demanded.

First of all, some semantic mutation operators for C need more information from the source code than traditional mutation operators. For example, in order to capture the misunderstandings of Floating-Point (FP) types in C, it is necessary to use the type information of variables and expressions but this generally is not included in a basic Abstract Syntax Tree (AST) [8]. Second, several scenarios in which SMT has particular value have been given [8]. One of these scenarios is porting of code because the same piece of code may result in different behaviours with different configurations. More specifically, for a pre-compiled programming language such as C, the executables generated from the same piece of source code may behave differently if they are compiled with different compilers, run on different operating systems or even compiled with the same compiler but different optimisation options. To investigate the semantic differences of a piece of C source code, the SMT tool should have the ability to be run on different configurations. However, there is generally a trade-off between portability and performance. For example, to achieve better performance, a number of MT tools adopted an approach that generates mutants from intermediate forms of the programming language [14], [15], [13]. This means the targeted source code is transformed into an intermediate form before conducting the MT analysis. For example, we considered building the SMT tool on CIL [16] by which a piece of C code is first transformed into CIL: a highly-structured, "clean" subset of C, and many advanced transformation and analysis features are then provided. However, there is a possibility that semantic discrepancies are introduced in the transformation from C code to CIL and then to executables. This is not ideal in terms of SMT. Therefore, an aim in designing the new tool was to avoid introducing possible semantic changes merely because of the SMT process.

As we decided to develop a new tool for the assessment of SMT, some other questions were also examined in the development process.

- Can we implement weak MT/SMT for C?
- Can we integrated SMT into the daily development routines of a C programmer?

We consider two aspects of the second question: is it possible to merge the SMT with other widely applied test routines; is it easy to integrated SMT of the C programming language

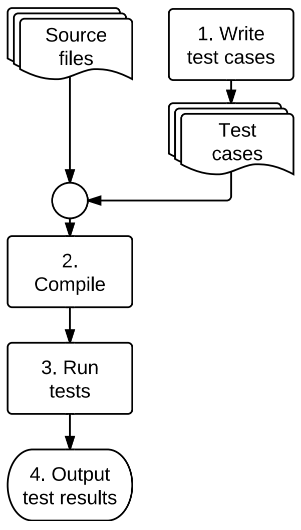


Figure 1. CFG of unit testing

into a popular Integrated Development Environment (IDE)?

Finally, we came up with **SMT-C** which is the first tool that does both traditional and weak SMT for C. It contains a set of modules including an Eclipse-based front end; uses *Check* [17] as a test harness and TXL [18] to generate the semantic mutants.

The remainder of this paper is organised as follows. In the next section, the Control Flow Graph (CFG) describing the process implemented in **SMT-C** is eventually introduced. The techniques used to implement important blocks in the **SMT-C** CFG in Figure 3 are then discussed in the following Sections III, IV and V. Section III describes the experiences of generating semantic mutants based on TXL. In Section IV, the advantages and issues of using *Check* to harness SMT are discussed. A GNU debugger (GDB) based implementation of Weak MT/SMT is then discussed (Section V). In Section VI, the Eclipse front-end of **SMT-C** is briefly introduced. Finally, we draw conclusions and discuss the future work.

II. CFGs and SMT-C

In this section, the overview of the implementation of **SMT-C** is discussed. A comparison between CFGs of unit testing, traditional MT and **SMT-C** is given to illustrate that **SMT-C**, supporting both strong and weak SMT, can be constructed by extending a general unit testing framework.

Unit testing is a process that checks the smallest testable parts of a program. For C, unit testing usually involves running tests on functions. A unit testing process generally contains four operations which are listed as follows.

- (*S1*) Unit test cases are normally hand coded by software engineers during the development process.
- (*S2*) Test cases are compiled into an executable together with the source files containing functions under test.
- (*S3*) Unit test is run by starting the executable.
- (*S4*) Generally, the executable should output information generated in the test process and log the test results.

The CFG of unit testing is given in Figure 1. using a Flowchart style. Rectangles with the same index refer to the same operation in all CFGs (Figures 1, 2 and 3). An index with a ' means that minor changes have been made to the operation with the same index and without '. For example, operation *S1* refers to the rectangle block with index 1. In a CFG with Flowchart style, rectangle blocks represent sequential operations and rectangle blocks with double-line edges represent predefined operations.

Since **SMT-C** adopted the syntactical approach to simulating the semantic misunderstandings, semantic mutation operators were developed to apply syntactic changes. Mutants were produced by applying semantic mutation operators on the targeted source code. The hypothesis of SMT is as follows. Let T, E and M represent a test suite, a type of semantic errors and a set of mutants generated by applying a semantic mutation operator which simulates E. T has a better ability to detect E if T can achieve a better mutation score against M. The mutation score of T is the percentage of mutants killed by T. From the control flow point of view, there is no difference between the SMT approach used in **SMT-C** and traditional MT.

For traditional MT, the original program and its mutants are distinguished if they produce a different output on the same test input [2], [19]. Or in other words, a test input *kills* a mutant M of the original program N if M and N produce different output when running with the test input. Generally, the meaning of killing a semantic mutant is the same as killing a traditional mutant. However, approximations may be introduced in giving verdicts for specific semantic mutations. For example, it may be necessary to approximate the equivalence between results of FP calculations since the semantics of FP calculation depend on a number of things including the actual expression, the underlying hardware, operating system, compiler, optimisation option used to generate the executable and so on [20]. For example, the source code in Listing 1 gave different results with different configurations. It printed "Unexpected result" on a Linux 32 bit system compiled with GCC v4.4.5 and optimisation level 0 and the difference between the two operands in the conditional expression is about $5.551121E-17$; it printed "Comparison succeeds" when the GCC optimisation level was changed to 3 or when running on a Linux 64 bit system compiled by GCC v4.3.5 with optimisation level 0.

Listing 1. Unexpected comparison results
```
float a = 10, b = 0.1;
if (a * b == 1)
  puts("Comparison succeeds");
else
  puts("Unexpected result");
```

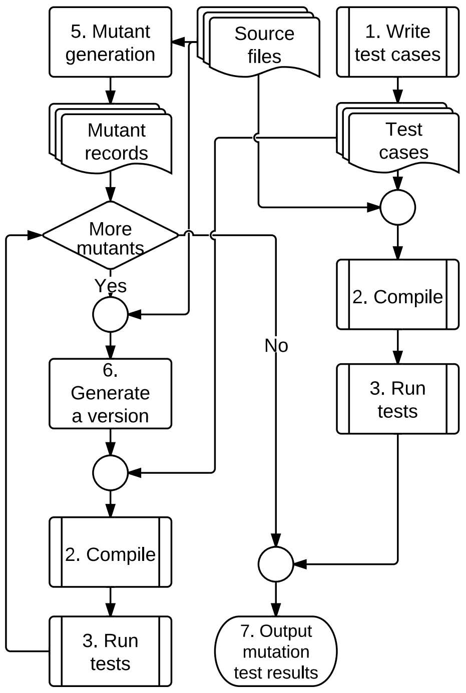


Figure 2. CFG of Traditional MT

From the general unit testing CFG in Figure 1, a traditional MT CFG can be synthesised with extra operations. Figure 2 shows a traditional MT control flow. We note that most operations of unit testing CFG are reused on the right-hand side of traditional CFG (*S1*→*S2*→*S3*→*S7*) because the output generated from mutants need to be compared with the output of the original program. To conduct MT, mutants should be generated from the original source code (*S5*). For each mutant, a new version of the program is synthesised (*S6*), compiled with the test cases (*S2*) and run (*S3*). After all mutants have been run with the given test cases, the results are compared with the test results generated from the original program. The mutation testing results are then parsed and output (*S7*). In addition to the operations in Figure 1, there are new operations in Figure 2: (*S5*) *mutant generation*, (*S6*) *generating a version* and (*S7*) *output MT results*. The CFG contains a loop around three operations: *S6*, *S2* and *S3* in which *S2* and *S3* are from the unit testing CFG. In other words, traditional MT repeats unit testing for each mutant and compares results with the original unit testing results.

Weak MT is an alternative to traditional MT (also called strong MT). Traditional MT compares the program output between the mutants and the original program. Weak MT, in terms of C, checks the changes of the program status immediately after the mutated expression or statement [21]. Therefore, it is desirable to implement weak MT in an interpretive way. However, this is difficult for a pre-compiled programming language as an independent interpreter may be required. For example, the weak MT module of Mothra was built on an interpreter of Mothra intermediate code of Fortran [14], [22]. To investigated the fault masking and equivalent semantic mutants problems, we were motivated to implement the weak SMT feature into **SMT-C**. To the best of our knowledge, there is not a weak MT tool supporting programming language C. To avoided developing an independent interpreter, GDB (7.0 onwards) was used as the interpreter in **SMT-C** since it can be controlled by python scripts in a batch mode [23]. The final CFG is thus given in Figure 3 in which weak MT/SMT for C is included. There are two extra operations, (*S8*) *generate debugger scripts* and (*S9*) *run test with GDB*. There is a block with index *S8'* because the scripts for running the original program and mutants are slightly different.

The CFG in Figure 3 was implemented in **SMT-C**. **SMT-C** was developed as a bundle of Eclipse platform so the control flow is encapsulated in Java classes. To seamlessly integrated with software engineers' daily routines, **SMT-C** was built upon Eclipse CDT (C/C++ Development Tooling) which is a fully functional C and C++ IDE of Eclipse. As we adopted TXL for the mutant generation and *Check* for the test harness, most of operations were developed as individual modules or delegated to third-party modules. The responsibilities of the Java code in the plugin are to build the control flow, prepare the input and parse the output of the individual modules. In the following sections, our experiences in the development of the important features of the **SMT-C** CFG are discussed in detail.

III. Semantic mutant generation

Adopting a syntactical approach, 19 semantic mutation operators were developed for C as shown in Table I. In addition, seven traditional mutation operators (including **OAAN**, **SCRB**, **SBRC**, **STRP**, **STRI**, **SSWM** and **SSDL**) described in [24] were also developed to provide a comparison with semantic mutation operators [8]. In the 19 semantic mutation operators, the first 13 operators were proposed in [20] based on existing industry experiences on the possible misunderstandings of the C programming language [25], [26], [27]. The remaining 6 FP comparison (FPC) operators in [20] were refined from two FPC operators implemented in [8]: MFC_E and MFC_R. Basically, all FPC operators find an FPC in a piece of source code and then change the expression to a function call, for example *fcmp* mentioned in Table I. The function being called introduces a *tolerance* in the comparison. Here, *tolerance* is a threshold which allows two FP numbers to be counted as equivalent if their difference is smaller than the threshold. The FPC operators is differentiated by the way that the tolerance is calculated [20].

The technique used for the development of these operators was TXL. It has been used for mutant generation before [11],

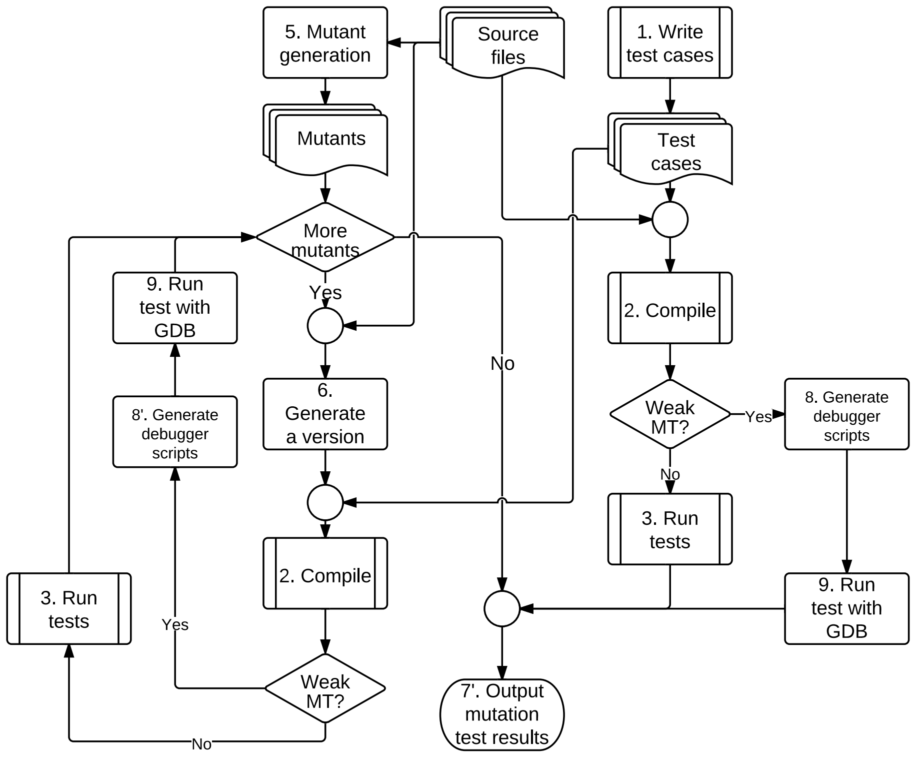


Figure 3. CFG with Weak MT/SMT

[28]. TXL is a general source transformation language with a long history [18]. A transformation process based on TXL has two steps. The source code is first parsed as an AST with its grammar and then a set of transformation rules written in TXL can be applied.

The main reason that we chose TXL to implement our semantic mutation operators is that TXL is easy and straightforward to use to make small changes to a piece of source code. For example, the logic of operator **SCRB**, which replaces jump statement 'continue;' with a 'break;', can be implemented using several lines of TXL code with the grammar given as follows [29]:

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

The other reason was that TXL is a general transformation tool which has the potential to support any form of description with a BackusNaur Form (BNF) style grammar. Refinement is one of the scenarios in which SMT has particular value [8]. Considering a process where software models are refined to source code, SMT may need to be applied to the models to capture the misunderstandings of the modelling language. We therefore intend to investigate SMT for models and other forms of software descriptions. Therefore, it appears likely that the TXL approach will be more flexible than those depending on parsers for specific languages since TXL applies to different description languages.

Although it is easy to implement TXL based mutation for most of the traditional C mutation operators and a number of C semantic mutation operators given in Table I such as **ASD** and **IMB**, the implementation of semantic mutation operators such as **DIA_F**, **FTA_F** and **MFC_E** was more complicated. For example, to implement **MFC_E** which replaces specific equivalence expressions by function calls, an expression is qualified for the transformation if it is first an equivalence expression and then the comparison has at least one FP number operand. The first condition can be implemented by searching the AST. However, to implement the second condition, the type information of operands of equivalence expressions needs to be inferred by applying a set of inference rules based on the AST. As a result, for such semantic mutation operators we developed a TXL type inference library for C which has more than 1600 LoC.

The other issues include a parsing problem and a formatting problem. With the given grammar [29], TXL parses most C source files. However, a small portion of C source files with complex macros cannot be parsed. This is because some strange C macros break the BNF grammar. In [20], we reported that 9 out of 598 C source files of the *nmath* library

Table I
SEMANTIC MUTATION OPERATORS

Operator	Description
	General semantic mutation operators [8]
AOR	replace '=' with '==' in conditional statements
ASD	remove additional semicolons after the condition expressions of *if* statements
LBC_I	add an *else* branch to an *if* statement without an *else* branch
LBM_I	modify the last *else if* branch of an *if* statement to an *else* branch
LBC_C	add a *default* branch to a *switch* statement without a *default* branch
LBM_C	modify the last *case* of a *switch* statement to a $default$ branch
MFC_E	mutate the FPC operators in an equality expression
MFC_R	mutate the FPC operators in a relational expression
DIA_F	mutate the results of division/modulus of integers using the floor method
DIA_T	mutate the results of division/modulus of integers using the tail method
FTA_F	FP type truncation adjustment using the floor method
FTA_T	FP type truncation adjustment using the tail method
IMB	inserting missing *break* statements into *switch* statements
	Semantic mutation operators for FPC [20]
MFC_C	introduce constant tolerance between comparison with single precision FP numbers
MFC_F	introduce tolerance using *fcmp* function between comparison with single precision FP numbers
MFC_H	introduce tolerance using a hybrid algorithm between comparison with single precision FP numbers
MDC_C	introduce constant tolerance between comparison with double precision FP numbers
MDC_F	introduce tolerance using *fcmp* function between comparison with double precision FP numbers
MDC_H	introduce tolerance using a hybrid algorithm between comparison with double precision FP numbers

of **R** software [30] were not parsed. The formatting problem of the TXL based mutant generation means that mutants generated by TXL are potentially formatted differently from the original file. This is because the TXL output follows the format defined in the grammar file. The properties of a mutant, such as the changed expression and the changed line number, are needed to run GDB based weak SMT and showing the modification made in a mutant in comparison view. A general *diff* program can be used to generate the properties, if the original source code and the mutant share the same format. Due to the formatting problem, it is thus impossible to use *diff* to generate real differences between a mutant produced by TXL and the original source code. In addition, instead of saving a mutant as a piece of slightly different source file, it is better to reserve the changes made to the original code [31]. This means that all mutants can be saved in one file with each mutant represented by a 'difference' record. This approach uses less storage and provides clear difference information.

SMT-C inherited the parsing problem from TXL. Initially, we did not repair it since this did not affect our research results [20]. However, the formatting problem has to be tackled as the difference information is important for displaying the changes of a mutant and generating weak MT/SMT scripts. The following approach was used to solve the problem. All pieces of source code are pre-formatted with TXL grammar before mutation operators are applied so that mutants and the original source code have the same format. A *diff* command can then be used to generate difference records for mutants.

IV. HARNESS TESTING USING UNIT TESTING FRAMEWORK

Unit testing is the earliest test for a piece of source code. The aim of unit testing is to improve the confidence in individual parts of a piece of software. In extreme programming, unit testing is the central activity and used as a contract that an individual part must satisfy. Traditionally, the harness of unit testing has generally been hand coded. Writing and running unit test cases are time consuming. Therefore, a number of unit testing frameworks have been implemented for simplifying the unit testing of C programs and are widely used in software industry [17], [32], [33].

MT was generally regarded as a unit testing technique, but most existing MT tools have their own styles of test harness [9], [10], [11], [12], [13]. This means that software engineers have to write new test cases and run MT in a specific style. As a result, the unit test cases generated with unit testing frameworks cannot be used in MT analysis. In addition, a C program generally contains multiple source files. Manually maintaining the configuration of such a program is time consuming. Therefore, configure files which automate maintenance routines are used to organise source files into a project, for example, *Autotools* [34] and Eclipse project files. To the best of our knowledge, there were no MT tools that support the project scale of MT, let alone benefit from the automatic features provided by the configuration tools.

To improve the usability of our MT/SMT tool and integrate it into the general software engineering process, we chose to implement MT/SMT operations based on a general unit testing framework. In this way, an MT/SMT process can be seamlessly integrated into the unit testing routines. The test harness of MT/SMT is delegated to the unit testing framework. Test cases for MT/SMT are written in the same style as the other unit test cases. Therefore, existing unit test cases can be reused for MT/SMT.

Specifically, *Check* was selected as the unit testing framework on which the MT/SMT operations of **SMT-C** is based. *Check* is developed based on Autotools which contains *Autoconf*, *Automake* and *Libtool*. *Autotools* is a configuration tool set which is popular in Free/Open source world. *Autotools* automates the troublesome configurations in development of cross-platform software, for example checking whether dependent libraries are installed and generating proper preprocessor directives which describe the underlying system for compilers. In addition, it provides a consistent and easy way for users to build, test and install the software. For example, to install software developed with *Autotools*, the installation generally uses the following commands.

```
./configure
make
make install
```

As a result, **SMT-C** was not implemented with its own test harness and project configuration. The loop structure (around *S6*, *S1* and *S2*) in Figure 2 shows that building the mutants and running tests are reused operation in the unit testing CFG given in Figure 1. As **SMT-C** supports the project scale mutation, before building such a mutant, *S6* (generate a version) is conducted: this refills the project source code directory with the mutated source code.

To use *Check* in Eclipse CDT, the Eclipse *Autotools* plugin is required. The Eclipse autotools plugin provides access to *Autotools* from Eclipse platform and also introduces a wizard to create a C project in Eclipse in *Autotools* style. **SMT-C** depends on the Eclipse *Autotools* plugin and only applies to the C projects created by the Eclipse *Autotools* plugin wizard. In an Eclipse environment with **SMT-C**, once a C project created for *Autotools* is activated, the menu provided by **SMT-C** is also activated. An MT/SMT process described in Figure 2 can then be started by clicking the corresponding menu item in the Eclipse menu.

Writing a *Check* test case is easy and intuitive. It is piece of code between a pair of predefined C macros as follows:

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

In addition to reusing the test harness of *Check*, our SMT tool also benefits from advanced features of *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. These advanced features make SMT-C an easy-to-use SMT tool in terms of test harness.

Check has many advantages, but the effort to learn it may not be justified for a small software project. To properly use *Check*, software engineers have to learn some pre-defined macros and APIs (Application Programming Interface). It is also required that software engineers are familiar with *Autotools* which may be difficult to learn for a new programmer. The other problem is that *Check* is an extension of *Autotools*, so it cannot be used independently, This is also true for **SMT-C** which can only be used with Eclipse with the *Autotools* plugin installed. We did not choose an independent C unit testing frameworks such as [33] since it was intended to develop **SMT-C** as an MT/SMT tool which could be closely embedded into software engineers' daily routines in a unified style.

V. Weak MT/SMT of C programming language

The idea behind weak MT/SMT is to check the program status immediately after the execution of the mutated component. When running the same test suite on mutant m, it is possible that m is weakly killed but not strongly killed. To investigate the difference between weakly and strongly killing of semantic mutants and fault masking, **SMT-C** was implemented to support Weak MT/SMT. To be more specific, **SMT-C** allows the status of a mutant and the original program to be compared after the first execution of the mutated statement. This type of weak MT is called *ST-WEAK/1* in [22].

Since weak MT checks the program status in the middle of executions, it is more convenient to develop weak MT based on an interpreter. This is because the interpreter can be controlled to check/output the program status naturally [22]. For the compiled programming language C, an intuitive way to automatically check the intermediate status of a program is to insert probing statements after each components in the original program and after the mutated component in mutants [35].

SMT-C used an approach that controls an interpreter to check/output intermediate program statuses. However, there is not a native interpreter for C, so an advance debugger, GDB, was used as the interpreter. The control flow of **SMT-C** with Weak MT/SMT is given in Figure 3. Once the option is set to run a Weak MT/SMT of the target function, the control flow takes branches in which python scripts are first generated (*S8* and *S8'*) and then the original program and mutants are executed by GDB (*S9*).

There are three issues in using GDB as the interpreter. Debugging with GDB is often a manually controlled process. However, recently, python scripting was introduced into GDB 7.0. Debugging of a C program can then be automated by a piece of python script. The other issue is that the accessible program statuses from GDB are limited by the debug information compiled into the executables. Therefore, a premise to use the GDB approach is to compile the executables with enough debug information. This generally means that the "-g" option should be used when compiling target functions. In terms of compiler options, it is also necessary to be careful about the choice of optimisation option "-O". This is because optimisation options may change the behaviour of resulted executables in specific situations. For example, it has been shown that the same piece of code generated different results when compiled with different optimisation options [20]. Finally, there is a performance issue when running programs with GDB. In our experience, running executables with GDB is about ten times slower than running the executable directly. However, this problem is less significant in SMT since semantic mutation operators generated fewer mutants on average [8].

It transpired that the *ST-WEAK/1* type of weak MT/SMT was not difficult to implement based on the scripting feature of GDB. Information regarding mutants is necessary for generating weak MT/SMT scripts (*S8* and *S8'*) such as the affected statement and the line number of the changed statement. Let us consider the process of using weak MT/SMT with a set of n mutants. Running the piece of script generated in *S8* should output results that can be compared with all output generated by n mutants in one run. This is achieved by inserting breakpoints into the original source code for all mutated lines of all n mutants. Generally, multiple breakpoints should be inserted but the number of breakpoints is usually less than n since different mutants may change the same statement. In an execution of the script with GDB, the program stops at the inserted breakpoints. Once the program stops, the current statement is executed with the *next* command of GDB and then the value of the left-hand side of the statement is logged or output. The script continues to run until the termination of the program. Running scripts in *S8'* generates the mutant output if the execution goes through the changed statement. The second piece of script just inserts a breakpoint at the mutated statement. Once the program stops at the breakpoint, the script logs or outputs the value of the left-hand side of the statement and exits the debugging process. Finally, the output from the mutants is compared with the output of the original program in block *Output mutation test results* (*S7'*).

The script for executing a mutant only contains a few lines of code. The main statements are as follows.

```
python
...
gdb.execute("handle SIGFPE nostop")
...
gdb.execute("run")
gdb.execute("next")
...
```

Some statements are skipped since they depend on the mutant information. For example, line 2 should be a statement that inserts a breakpoint at the line where the change is made to generate the mutant. The position of the breakpoint changes for different mutants. Let us suppose that a breakpoint has been set up before line 4. In the piece of script, the first line is to start python scripting in GDB batch mode; line 3 tells GDB not to terminate when an error signal raised by an erroneous arithmetic operation is received; lines 5 and 6 run the program and execute the current statement. Line 3 is necessary since by default GDB terminates the debugging process even when receiving a benign system signal. However, this often is not true in a real execution. The effect of line 5 is that the program stops at the breakpoint which should have been set up at line 3 if the test input triggers a path which goes through the mutated statement. After the execution of line 6, there are some statements to record the value of the left-hand side of the mutated statement.

Another advantage of using GDB is that it is independent of compilers and languages. Therefore, it is possible to use the same approach to run weak MT/SMT to check the semantic misunderstandings introduced by different compilers (Intel C compiler, Microsoft C compiler etc.) and on different programming languages including Ada, C, C++, Objective-C, Pascal and Java which are supported by GDB. This improves the portability of the tool and lead to a more flexible architecture.

VI. Eclipse based front-end

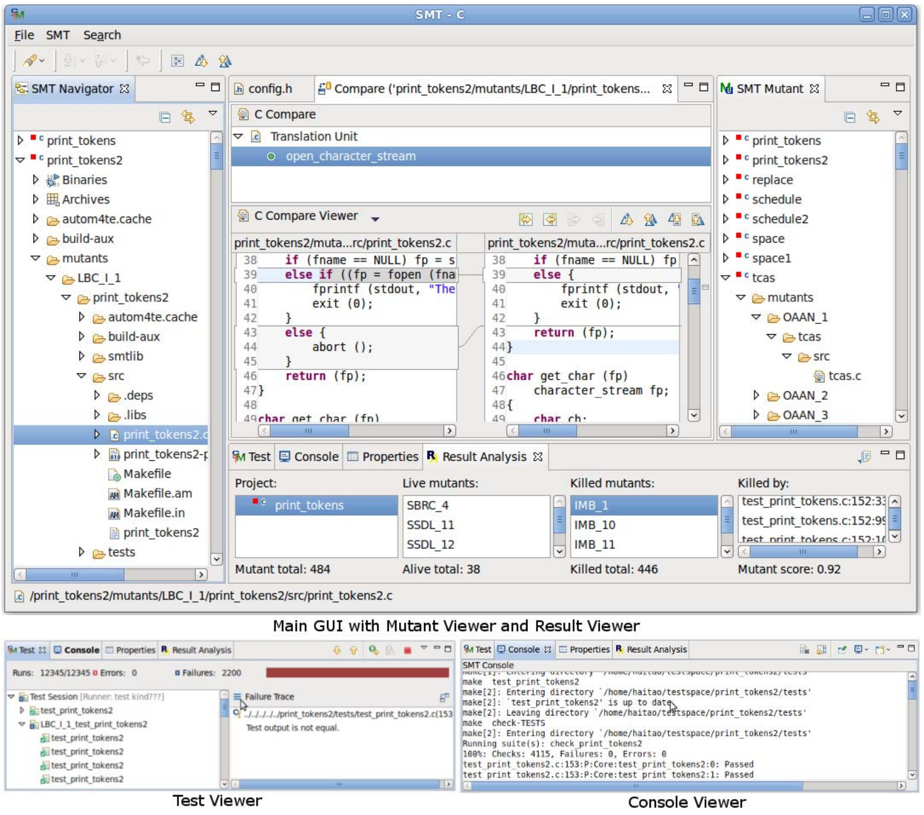
Sections III, IV and V described the most important operations in the CFG given in Figure 3. The individual operations were initially implemented as executables or command lines calling third-party software. For example, generating mutants from a given source file, *foo.c*, with mutation operator **DIA_F**, can be achieved as follows:

```
txl foo.c op.dia_f.txt -o mut_foo.c
```

In this command, *op.dia_f.txt* is the TXL transformation rule which generates mutant *mut_foo.c* from *foo.c*. With all individual operations in Figure 3 implemented, functionalities provided by them need to be presented to software engineers with a unified interface. In this section, the approach used to implement the interface of **SMT-C** is discussed.

A number of approaches can be chosen to automate SMT control flow. For example, shell scripts based commands is a good choice which is light-weighted and flexible. In the development of **SMT-C**, we used a bottom-up approach. With the components being prepared, to check whether they can work together, we used shell scripts to test the integrated functions. Actually, MT/SMT analysis can still be run manually by calling these test scripts with the modules of **SMT-C**. However, a GUI based approach can be more user-friendly, especially for new programmers. For example, a GUI approach saves efforts on memorising the individual commands and their options. Other than implementing the CFG given in Figure 3, the front-end of the tool should also output useful information generated by each individual operation including the mutated statement of a mutant, test cases, original test results, and mutation test results. It can be easier to present these pieces of information at the same time using a GUI approach. Therefore, a GUI front-end was developed for **SMT-C**.

As mentioned earlier, **SMT-C** was designed to be embedded into daily routines of a software engineer. As many C programmers use Eclipse CDT as their IDE and Eclipse CDT supports *Autotools*, *Check*, and *diff* for comparison between mutants and the original program, it was chosen as the basis of the front-end of **SMT-C**. The front-end is thus



```
    if (fname == NULL) fp = s
    else if ((fp = fopen (fna
        fprintf (stdout, "The
        exit (0);
    }
    else {
        abort ();
    }
    return (fp);
}

char get_char (fp)
```

```
    if (fname == NULL) fp
    else {
        fprintf (stdout, '
        exit (0);
    }
    return (fp);
}

char get_char (fp)
    character_stream fp;
{
    char ch;
```

Project: print_tokens
Live mutants: SBRC_4, SSDL_11, SSDL_12
Killed mutants: IMB_1, IMB_10, IMB_11
Killed by: test_print_tokens.c:152:3, test_print_tokens.c:152:9
Mutant total: 484 Alive total: 38 Killed total: 446 Mutant score: 0.92

/print_tokens2/mutants/LBC_I_1/print_tokens2/src/print_tokens2.c

Runs: 12345/12345 Errors: 0 Failures: 2200

Failure Trace
../../../../print_tokens2/tests/test_print_tokens2.c(153
Test output is not equal.

SMT Console
```
make  test_print_tokens2
make[2]: Entering directory `/home/haitao/testspace/print_tokens2/tests'
make[2]: `test_print_tokens2' is up to date.
make[2]: Leaving directory `/home/haitao/testspace/print_tokens2/tests'
make  check-TESTS
make[2]: Entering directory `/home/haitao/testspace/print_tokens2/tests'
Running suite(s): check_print_tokens2
100%: Checks: 4115, Failures: 0, Errors: 0
test_print_tokens2.c:153:P:Core:test_print_tokens2:0: Passed
test_print_tokens2.c:153:P:Core:test_print_tokens2:1: Passed
```

Figure 4. GUI of **SMT-C**

developed as a plugin of Eclipse and written in Java. Several *classes* were used to encapsulate the individual modules and then these *classes* were assigned to individual handler *classes* which can be attached to menu items and buttons in the Eclipse main menu and viewers[1]. Above the *classes* for encapsulating individual functionalities, there are two *classes* which implement control flows of traditional and weak MT/SMT, respectively.

To display the relevant information, **SMT-C** has four viewers: the mutant viewer, the test viewer, the results viewer and the console viewer. The upper part of Figure 4 shows the main window of **SMT-C**. The mutant viewer is on the right-hand side of the main window in which the generated mutants can be managed. The results viewer is used to display the results of SMT including killed mutants, mutation score and so on. The test viewer and the console viewer are displayed separately below the main window. Software engineers can use the test viewer to start SMT and view the results of each test case that has been run. The console viewer provides raw information generated by individual commands. High-level features of Eclipse and CDT have been reused in **SMT-C**. For example, the middle of the main window of **SMT-C** in Figure 4 shows two mutants being compared and is implemented reusing the *diff* module of Eclipse.

[1]Viewers are small windows in Eclipse GUI exclude the editor window.

VII. Conclusions and Future Work

In this paper, we discussed our experiences regarding the development of an SMT tool for C: **SMT-C**. In addition to providing basic functionalities, we also tried to achieve the following goals: Weak MT/SMT in C; good portability; embedded into the daily routines of a C programmer; an easy-to-use front-end. Although **SMT-C** is still under construction, with the experiences we have, we argue that the approach adopted in **SMT-C** is a promising method for these goals.

For generating semantic mutants, we used TXL which is a high-level transformation tool. The TXL approach has no limitations on platform, compiler and specific language format. This ensures that the mutant generation in **SMT-C** is highly portable. In addition it applies to many other languages including modelling languages. This flexibility is important since we plan to investigate SMT for different types of abstract description of software.

In designing the test harness of **SMT-C**, we used an approach based on a unit testing framework *Check* which is an extension of *Autotools*. Two main goals were achieved. *Autotools* enabled our tool to support a project scope MT/SMT; in contrast most other mutation tools were developed for single source file programs. The other is that the use of *Check* enables the seamless merging of unit testing and SMT. As a result, given a set of *Check* style test cases, SMT can be started by clicking one menu item.

To implement weak MT/SMT for C, we used an interpretive approach. As C is a pre-compiled programming language, there is no native C interpreter. The most recent version of GDB was used as an interpreter. The advantage is that it is easier than developing an independent interpreter and potentially has better portability. However, this approach has a performance issue. Despite this performance issue, it satisfied our needs for analysis of *ST-WEAK/1* style of SMT.

Finally, an Eclipse based front-end was provided for organising the functionalities of **SMT-C** and presenting the useful information generated in the MT/SMT analysis. This approach provided an easy to use interface to those already familiar with Eclipse IDE. In addition, it should facilitate the integration of the MT/SMT into the daily routines of developers.

We plan to make **SMT-C** open to download in the near future.[2] Before that, we will further refine the existing functionalities, for example improving the way that mutants are stored and implementing more functionalities such as the other types of weak MT/SMT. Although, the performance of **SMT-C** was not a significant problem for the experiments that we have conducted, there is still a long way to reach the industry standard. For example, **SMT-C** took hours to finish the SMT experiments in [8]. The performance of the tool should be carefully evaluated and then improved.

Acknowledgements

This research was partially supported by the UK EPSRC project EP/G04354X/1, The Birth, Life and Death of Semantic Mutants.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of 27th International Conference on Software Engineering*, St. Louis, USA, 2005, pp. 402–411.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practical programmer," *IEEE Computer*, vol. 11, no. 4, pp. 31–41, 1978.

[3] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: An experimental comparison of effectiveness," *Journal of Systems Software*, vol. 38, no. 3, pp. 235–253, 1997.

[4] R. M. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Journal of Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 233–262, 1999.

[2]For the time being, please send emails to haitao.dan@brunel.ac.uk for further information about **SMT-C**.

[5] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering Methodology*, vol. 1, no. 1, pp. 3–18, 1992.

[6] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, pp. 649–678, 2011.

[7] J. A. Clark, H. Dan, and R. M. Hierons, "Semantic mutation testing," in *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops*, April 2010, pp. 100 –109.

[8] ——, "Semantic mutation testing," *Science of Computer Programming*, vol. In Press, Corrected Proof, 2011.

[9] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, 2001.

[10] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'93)*, Cambridge, Massachusetts, 1993, pp. 139–148.

[11] J. S. Bradbury, J. R. Cordy, and J. Dingel, "ExMAn: A generic and customizable framework for experimental mutation analysis," in *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, Raleigh, North Carolina, November 2006, pp. 57–62.

[12] M. Ellims, D. C. Ince, and M. Petre, "The Csaw C mutation tool: Initial results," in *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, Windsor, UK, 10-14 September 2007, pp. 185–192.

[13] Y. Jia and M. Harman, "MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language," in *Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08)*, Windsor, UK, 29-31 August 2008, pp. 94–98.

[14] A. J. Offutt and K. N. King, "A Fortran 77 interpreter for mutation analysis," in *Proceedings of the Symposium on Interpreters and interpretive techniques*, ser. SIGPLAN '87, St. Paul, USA, 1987, pp. 177–188.

[15] Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon, "MuJava: a mutation system for java," in *Proceedings of the 28th international Conference on Software Engineering (ICSE '06)*, Shanghai, China, 20-28 May 2006, pp. 827–830.

[16] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *Compiler Construction*, ser. Lecture Notes in Computer Science, R. Horspool, Ed. Springer Berlin / Heidelberg, 2002, vol. 2304, pp. 209–265.

[17] "Check: A unit testing framework for C," http://check.sourceforge.net/, Accessed in 2011.

[18] J. Cordy, "TXL-a language for programming language tools and applications," *Electronic notes in theoretical computer science*, vol. 110, pp. 3–31, 2004.

[19] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, pp. 279–290, 1977.

[20] H. Dan and R. M. Hierons, "Semantic mutation analysis of floating-point comparison," in *the 5th International Conference on Software Testing, Verification and Validation, Accepted*, Montreal, Canada, 2012.

[21] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, vol. 8, no. 4, pp. 371–379, 1982.

[22] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 337–344, 1994.

[23] "GDB: The GNU Project Debugger," http://www.gnu.org/software/gdb/, Accessed in 2011.

[24] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. Martin, A. Mathur, and E. Spafford, "Design of mutant operators for the C programming language," Department of Computer Sciences, Purdue University, Tech. Rep., 1989.

[25] A. Koenig, *C traps and pitfalls*. Addison Wesley, 1989.

[26] L. Hatton, *Safer C: Developing Software in High-integrity and Safety-critical Systems*. McGraw–Hill, 1994.

[27] R. Seacord, *Secure Coding in C and C++*. Addison-Wesley Professional, 2005.

[28] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *Proceedings of the 2nd Workshop on Mutation Analysis*, Nov. 2006, pp. 83–92.

[29] J. R. Cordy, A. J. Malton, and C. Dahn, "TXL GNU C grammar," http://www.txl.ca/nresources.html, Accessed in 2011.

[30] R. Gentleman, R. Ihaka *et al.*, "The R project for statistical computing," http://www.r-project.org, Accessed in 2011.

[31] K. N. King and A. J. Offutt, "A Fortran language system for mutation-based software testing," *Software– Practice and Experience*, vol. 21, no. 7, pp. 685–718, Jul. 1991.

[32] K. Sutou, H. Ikezoe, and Y. Hayamizu, "Cutter: A Unit Testing Framework for C and C++," http://cutter.sourceforge.net/, Accessed in 2011.

[33] A. Kumar and J. St.Clair, "CUnit: A Unit Testing Framework for C," http://cunit.sourceforge.net/, Accessed in 2011.

[34] "Autoconf," http://www.gnu.org/software/autoconf/, July 2010.

[35] M. R. Girgis and M. R. Woodward, "An integrated system for program testing using weak mutation and data flow analysis," in *Proceedings of the 8th International Conference on Software Engineering (ICSE'85)*, London, England, August 1985, pp. 313–319.