Mutation Testing Applied to Validate SDL Specifications

Tatiana Sugeta[1], José Carlos Maldonado[1] and W. Eric Wong[2]

[1] Universidade de São Paulo,
Instituto de Ciências Matemáticas e de Computacão,
São Carlos, São Paulo, Brazil,
P.O. Box 668, ZIP Code 13560-970,
{tatiana, jcmaldon}@icmc.usp.br
[2] University of Texas at Dallas,
Department of Computer Science,
Richardson, TX, USA,
ZIP Code 75083,
ewong@utdallas.edu

Abstract. Mutation Testing is an error-based criterion that provides mechanisms to evaluate the quality of a test set and/or to generate test sets. This criterion, originally proposed to program testing, has also been applied to specification testing. In this paper, we propose the application of Mutation Testing for testing SDL specifications. We define a mutant operator set for SDL that intends to model errors related to the behavioral aspect of the processes, the communication among processes, the structure of the specification and some intrinsic characteristics of SDL. A testing strategy to apply the mutant operators to test SDL specifications is proposed. We illustrate our approach using the Alternating-Bit protocol.

Keywords: Specification Testing, Mutation Testing, SDL.

1 Introduction

Testing is an important activity to guarantee the quality and the reliability of a product under development. The main objective of testing activity is to identify errors that were not yet discovered in software products. To obtain a reliable software, specification testing is as important as program testing since the earlier the errors are detected in the life cycle the less onerous is the process to remove them. The success of the testing activity depends on the quality of a test set. The quality of the testing activity is by itself an issue in the software development process. One way to evaluate the quality of a test case set `T` is to use coverage measures based on testing criteria.

Mutation Testing is a criterion initially proposed to program testing but some works have shown this criterion can also be applied to specification testing and conformance testing [6, 15–18, 20, 25, 26, 29]. Mutation Testing consists of

generating mutants of the program/specification, based on a mutant operator set that intents to model common, typical errors made during the development. The objective is to select test cases that are capable to distinguish the behavior of the mutants from the behavior of the original program/specification. Mutation Testing provides mechanisms to evaluate the quality of a test set and/or to generate test sets [11]. Mutation Testing has also been explored to support Integration Testing [10, 19]. Delamaro et al. [10] proposed Interface Mutation to explore interface errors related to the connections among program units, in a pairwise approach, and Ghosh and Mathur [19] proposed Interface Mutation to explore errors in the interface of components in a distributed application.

Formal techniques have been used to specify safety critical systems like bank control, air traffic control, metro control, patient hospital monitoring and communication protocols. Examples of these techniques are Statecharts, Petri Nets, Estelle and SDL. SDL (Specification and Description Language) is a language standardized and maintained by ITU-T (*International Telecommunications Union*) for the specification and description of telecommunications systems. Although this initial intention, SDL has been used to describe reactive systems such as real-time, event-driven and communicating systems. The behavior of a system modelled by SDL is described by processes, that behave like Communicating Extended Finite State Machines (CEFSMs).

Different techniques have been proposed to generate test cases from the SDL specifications [7, 20, 21, 32]. These techniques are applied to the conformance testing, that uses the test set generated to test the implementation of the system. In conformance testing it is supposed the specification is correct. An usual formal verification technique applied to guarantee the correctness of the specification is model checking. Model checking is based on state exploration and allows to verify some software properties. Although by using model checking some information of the structural coverage can be obtained, this technique do not stress to provide evidences about how much the specification was tested that could provide a quantitative measure about the testing being executed.

Motivated by previous researches that have investigated Mutation Testing to validate specifications based on formal techniques such as Finite State Machines [14, 16], Statecharts [18, 30], Petri Nets [17, 27, 28], Estelle [26, 29], in this paper we propose the Mutation Testing to test specifications written using SDL. We present a mutant operators set and a mutation-based testing strategy to guide the tester to apply the mutant operators and to explore the behavioral aspect, the communication among the processes and the structure of the SDL specification. To illustrate our definitions we use the well known Alternating-Bit protocol [31].

This paper is organized as follow: Section 2 contains an overview of some related works. In Section 3 we present an overview of basic concepts of SDL. The main concepts related to Mutation Testing are discussed in Section 4. The mutant operators set and a proposed Incremental Testing Strategy are presented in Section 5. In that section we also illustrate examples of application of some mutant operators. Our final comments are discussed in Section 6.

2 Related Work

Probert and Guo [26] proposed a technique to test Estelle specifications based on mutation, named E-MPT (Estelle-directed Mutation-based Protocol Testing). This technique validates the EFSMs defined in the specification. It generates the mutants from the specification and translates, using an Estelle compiler, the original specification and its mutants to C programs. The codes generated by the compiler are not completed and they need to be completed by the tester. The C programs generated are executed and the obtained results are compared.

Bousquet et al. [6] applied Mutation Testing in the conformance testing. In their experiment, the criterion is used to verify if the implementation of a conference protocol is according to its specification. The specification was written using Lotos. The test cases are generated based on the Lotos specification and the implementation is tested when using this test case set generated.

Ammann and Black use Mutation Testing and model checking to automatically produce tests from formal specifications [3] and measure test coverage [2]. The system is specified by Finite State Machines. To the former, each transition of the state machine is represented as a clause in temporal logic. To generate tests, the mutant operators are applied to all temporal logic clauses, resulting in a set of mutant clauses. The model checker compares the original state machine specification with the mutants. When an inconsistent clause is found, the model checker produces a counterexample if possible, and it is converted to a test case. To measure the coverage of a test set, each test is turned into a finite state machine that represents only the execution sequence of that test. Each state machine is compared by the model checker with the set of mutants produced previously. A mutation adequacy coverage metric is the number of mutants killed divided by the total number of mutants. Black et al. [5] refined the mutant operators set defined in Ammann et al. [3] and proposed new ones to be applied in the same approach that combines mutation testing and model checking.

Kovács et al. [20] have used Mutation Testing to generate and select test cases to be applied at the conformance testing of communication protocols specified using SDL. The test cases generated at specification level are used to test the programs implemented based on the specification and its mutants. Two algorithms were proposed to select test cases. The first one has an SDL specification as input and the result is a test case set that is Mutation Testing-adequate. The second one has two inputs, an SDL specification and a finite test case set. This algorithm analyzes the initial test case set and only those ones that identify the mutants are selected. A tool was implemented using Java to automate the second algorithm.

Fabbri et al. [15–18] have explored Mutation Testing at the specification level, analyzing the adequacy of this criterion on testing the behavioral aspects of reactive systems specified using Finite State Machines [16], Statecharts [18] and Petri Nets [17]. Fabbri et al. defined mutant operators for these three formal techniques. Mutants are generated by applying the mutant operators to the specification being tested. The mutant operators set models the more typical errors related to the specification technique in use. After the mutants generation,

each mutant is simulated and the results are compared to the results of the original specification. These steps contribute to the analysis of the mutation testing adequacy.

Souza et al. [29] investigated the application of Mutation Testing to validate Estelle specifications, exploring with this criterion the behavioral aspect, the communication among the modules and the structure of the specification. Souza et. al present a more complete mutant operators set than the one proposed by Probert and Guo [26], considering aspects such as the interface among modules, the hierarchical structure and the parallelism of the specification.

Fabbri et al. [18] and Souza et al. [29] established Incremental Testing Strategies to aid the application of Mutation Testing to Statecharts and Estelle, respectively. By using these strategies it is possible to prioritize some specific aspects, according to the features the tester wants to explore in the testing activity.

3 SDL: Overview

SDL is a language standardized and maintained by ITU-T (*International Telecommunications Union*) for the specification and description of telecommunications systems. Its first version is from 1976 and since then the language has been modified and improved to be as complete as possible. The newest version is SDL 2000. Although SDL was initially proposed to telecommunications systems, it has been used to describe reactive systems such as real-time, event-driven and communicating systems.

An SDL specification consists of a system, blocks, processes and channels. An SDL system specification is compounded by blocks, which exchange messages or signals with each other and the environment through the channels. The blocks can be decomposed recursively into sub-blocks and in the last level of this decomposition are the processes. Communication between processes is asynchronous and is also through channels. Channels can be uni or bi-directional. All the signals received by a process are merged into the individual process First In First Out (FIFO) queue, in the order of their arrivals.

The behavior of an SDL system specification is described by processes, that behave like Communicating Extended Finite State Machines. A process consists of a set of states and transitions that connect the states. When in a state, a process initiates a transition by consuming an expected signal from its input queue. Non expected signals in a state are implicitly consumed, that means to discard them and remains in the same state. Sometimes, it can be interesting to keep a signal in the queue to be consumed later by the process. In this case, the *save* construction can be used and just change the order of signals consumption by a process. Consuming a signal can result in another signals and update in the variables values.

SDL process can access the global timer using *now* which returns the current time. A timer is set to expire after a certain time from the current time defined by *now*. When a timer expires it sends a signal with its name to the process and

this signal arrives at the input queue of the process. After the timer signal is consumed the timer is reset.

4 Mutation Testing

Mutation Testing is used to increase the confidence that a software product `P` is correct by producing, through small syntactic changes, a set of mutant elements that are similar to `P`, and creating test cases that are capable of causing behavioral differences between `P` and each one of its mutants. These changes are based on an operators set called mutant operators. To each operator it is associated an error type or an error class that we want to reveal in `P`.

The definition of mutant operators is a crucial factor for the success of Mutation Testing. Very simple operators are usually defined based on the competent programmer hypothesis, which states that a program produced by a competent programmer is either correct or near correct. The tester must construct test cases that show that these transformations lead to incorrect programs. Another hypothesis considered by Mutation Testing is the coupling effect that, according to DeMillo et al. [12], can be described as "test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it would also implicitly distinguish more complex errors".

Mutation Testing consists of four steps: mutant generation; execution of the program `P` based on a defined test case set `T`; mutant execution; and adequacy analysis. All the mutants are executed using a given input test case set. If a mutant `M` presents results different from `P`, it is said to be dead, otherwise, it is said to be alive. In this case, either there are no test cases in `T` that are capable to distinguish `M` from `P` or `M` and `P` are equivalent, that means they have the same behavior (or output) for any data of the input domain. The objective must be to find a test case set `T` to kill all non-equivalent mutants; in this case `T` is considered adequate to test `P`.

DeMillo [11] notes that Mutation Testing provides an objective measure for the confidence level of the test case set adequacy. The Mutation Score, obtained by the relation between the number of mutants killed and the total number of non-equivalent mutants generated, allows the evaluation of the adequacy of the test case set used and therefore of the program under testing. A test case set `T` is adequate to a program `P` with respect to mutation testing coverage if the mutation score is 1.

The computational cost can be an obstacle to the use of Mutation Testing due to the high number of mutants generated and to be analyzed so that the equivalent ones can be identified. Since, in general, the equivalence is an undecidable question the equivalent mutants are interactively obtained by the tester. Some alternatives were proposed to program testing [4, 23, 34]. Wong et al. [34] provide evidences that examining only a small percentage of the mutants may be an useful heuristic for evaluating the test sets. Offutt et al. [23] and Barbosa et al. [4] have proposed the use of an essential operator set, so that a high mutation score against this essential set would also determine a high mutation

score against the full set of mutant operators. Another option is to automatically generate test cases and determine equivalent mutants [13, 24]. Simão and Maldonado [27] proposed an algorithm to generate test cases based on Mutation Testing to validate Petri Nets. Although it is undecidable, in some cases equivalent mutants can be identified using this algorithm.

Although Mutation Testing, as mentioned before, was proposed for program testing, it can be of help for validating a specification even considering that there is no mutation that would "restore" the correct behavior of the specification. The mutations can lead the tester to an error-revealing mutant without leading to the correct behavior. If k-mutants (more than a single change in the mutant) are considered it can be argued that would exist a mutant that presents the correct behavior but always depending on the quality of the specification under test. By defining a set of mutant operators and considering just a single change in each mutant, in fact, the space of possible wrong specifications has been reduced, reducing the cost of the Mutation Testing and assuming that single errors would lead to discover multiple and more complex errors. This assumption has to be explored in further studies.

5 Mutation Testing applied to SDL

Earlier researches on testing specifications using Mutation Testing indicate that this criterion may contribute to the improvement of these activities [16, 26, 29], since it can complement other testing methods. This fact motivates the analysis of the adequacy of Mutation Testing in the context of SDL specifications.

As commented before, the definition of mutant operators is a key factor for the success of this criterion. Like Fabbri et al. [15, 16, 17, 18] and Souza et al. [29], we propose a mutant operators set for SDL based on some previous works: the control structure sequencing error classes defined by Chow [8], the mutant operators for boolean expressions defined by Weyuker et al. [33] and the mutant operator set for C language defined by Agrawal [1], Delamaro and Maldonado [9], Delamaro et al. [10]. Added to them, we explore some intrinsics features of SDL like *save* and *task* commands.

To illustrate the application of Mutation Testing in an SDL specification we use the well known Alternating-Bit protocol, which is a simple form of the "sliding window protocol" with a window size of 1 [31]. This protocol provides a reliable communication over a non-reliable network service using a one-bit sequence number (which alternates between 0 and 1) in each message or acknowledgement to determine when messages must be retransmitted. This protocol is composed of a sender and a receiver processes that communicate through two channels (Medium1 and Medium2). Figure 1 illustrates the Alternating-Bit protocol at the system level, with three blocks: sender_block, medium and sender_block.

The mutant operators are divided into three different classes: Process Mutant Operators, Interface Mutant Operators and Structure Mutant Operators. Table 1 illustrates the set of mutant operators defined for SDL and the number of mutants generated by them for the Alternating-Bit protocol. All of the opera-

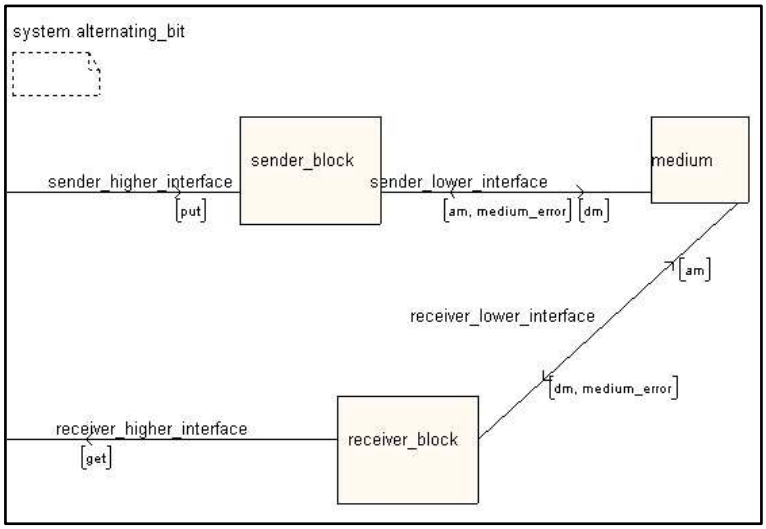

Fig. 1. System Level of the Alternating-Bit Protocol Specification in SDL

tors generated 261 mutants: 77, 51 and 133 by the Process Operators, Interface Operators and Structure Operators, respectively. Some operators do not generate any mutant since the syntactic structures that these operators act on do not occur in this protocol specification.

Given a specification S, a mutant set of S is generated, $\phi(S)$. A test set T is adequate for S with relation to $\phi(S)$ if for each specification Z of $\phi(S)$, either Z is equivalent to S, and in this case Z and S have the same behavior for T, or Z differs from S at least on a test point. To distinguish the mutant behavior from the original one, we analyze the final states of all processes reached after the execution with the test case set. Considering s a statement in a specification S and s_m the same statement but containing some mutation to generate the mutant Z. Three conditions must be satisfied by a test case T to distinguish Z from S [13]:

1. Reachability: s_m must be executed.
2. Necessity: The state of the mutant Z immediately after some execution of s_m must be different from the state of the original specification S after the execution of s.
3. Sufficiency: The difference in the states of S and Z immediately following the execution of s_m and s must be propagated until the end of execution of S or Z so that the final states reached by them when executed with T are different.

For SDL we consider that a typical test sequence is constituted by the sequence of signals exchanged during the execution of the specification. For example, a possible test sequence for the Alternating-Bit protocol is $ts = <put(m),$

Table 1. Mutant Operators for SDL

Process Mutant Operators			
1. Origin State Replacement	03	15. Coverage of Code	15
2. State Definition Exchanged	01	16. Question of Decision Negation	04
3. Destination State Replacement	09	17. Answer of Decision Exchanged	04
4. State Missing	02	18. Answer of Decision Missing	08
5. Transition Missing	08	19. Stop Process Missing	0
6. Condition Missing	0	20. Save Missing	01
7. Negation of Condition	0	21. Signal Saved Missing	0
8. Boolean Assignment Replacement	0	22. Signal Saved Replacement	0
9. Variable by Variable Replacement	12	23. SET TIMER Missing	0
10. Variable by Constant Replacement	0	24. RESET TIMER Missing	0
11. Variables/Constants Increment/Decrement	0	25. CREATE Process Missing	0
12. Unary Operator Inclusion in Variables	0	26. Arithmetic Operator Replacement	0
13. Task Replacement	0	27. JOIN/LABEL Replacement	0
14. Task Missing	02	28. Relational Operator Replacement	08
TOTAL			77
Interface Mutant Operators			
Group I: Calling Point		Group II: Called Process	
1. Output Missing	11	9. Interface Variables Replacement	08
2. Output Replacement	15	10. Non-Interface Variable Replacement	0
3. Output Destination Replacement	0	11. Variable Increment/Decrement	0
4. Signal Route of Output Replacement	06	12. Unary Operators Inclusion in Variable	0
5. Parameters Replacement	0	13. Boolean Assignment Replacement	0
6. Parameters Increment/Decrement	0	14. Input Missing	07
7. Order of the parameters Exchanged	0	15. Input Replacement	04
8. Unary Operators Inclusion in Parameters	0		
TOTAL			51
Structure Mutant Operators			
1. Signal of Signal List Inclusion			44
2. Signal of Signal List Missing			12
3. Signal of Signal List Exchanged			35
4. Signal routes/Channels Missing			16
5. Connection between Channels and Signal routes Missing			08
6. Connected Channels/Signal routes Replacement			18
TOTAL			133

dm(m,0), dm(m,0), get(m) ∧ am(0), am(0)>. This sequence corresponds to the following steps: Sender sends a message to Receiver ($put(m)$ signal). The message (m) is packed and it is sent to the Medium1 ($dm(m, 0)$) with an identifier bit (0). Medium1 sends the message to Receiver ($dm(m, 0)$). The message arrives to Receiver that verifies that it is the message it was expecting, stores the message ($get(m)$) and sends the acknowledgement to Sender ($am(0)$) through Medium2. Medium2 receives the acknowledgment and sends it to Sender ($am(0)$). The acknowledgment is received by the Sender and it is recognized as the expected acknowledgment.

For the test sequence ts, the final states activated are $sa = <$ *([(wait_put)], [(wait_dm), (wait_am)], [(wait_dm)])* >, considering the following order of processes: ([Sender], [Medium1,Medium2], [Receiver]).

In the following we describe the three classes of mutant operators and present an informal definition of one mutant operator for each class. For each one of these mutant operators we illustrate one mutant generated and execute them with the test sequence ts.

5.1 Process Mutant Operators

The operators of this class model errors related to the behavior of processes that is similar to the Communicating Extended Finite State Machines (CEFSMs) behavior. To define this class of operators, we consider the particular features of SDL and two previous works: the set of mutant operators to Extended Finite State Machines (EFSMs) defined by Fabbri et al. [18] and the set of operators defined by Souza et al. [29] to explore mutation on modules of Estelle specifications that behave like EFSMs. Our mutant operators set models: transitions and states errors; expressions, mathematic operators, variables and constants errors, and; timers errors (set and reset).

– Example: Destination State Exchanged
  This operator models state errors by exchanging the destination of each transition. The state defined at the *nextstate* command is mutated by other states defined in the same process and by the `-` symbol.
  This operator is applied to the Sender process of the Alternating-Bit protocol. The transition fired by `am(j)` event when in the `wait_am` state has as destination state `wait_put`. To generate the mutants this state is replaced by the other state in the process, that is `wait_am`, and by the - symbol. Figure 2 illustrates the part of the original specification where one mutation is done and one of the mutants generated.
  When this mutant is executed with ts the final states activated are $sa = <$ *([wait_am], [wait_dm, wait_am], [wait_dm])* $>$. The `wait_am` state in Sender process is different from the expected one which was the `wait_put` state. As a result, the mutant is distinguished from the original specification and so is considered dead. In other case, if the mutant is executed with $ts_i =$ *<put(m), dm(m,0), dm(m,0), get(m)* $\wedge$ *am(0), medium_error>*, the final states activated are $sa = <$ *([wait_am], [wait_dm, wait_am], [wait_dm])* $>$. The same final states are activated when the original specification is executed with ts_i. Thus, the test case ts_i is not able to distinguish this mutant from the original specification.

Original Specification	**Mutant Specification**
STATE wait_am;	STATE wait_am;
INPUT am (j);	INPUT am (j);
DECISION j = i;	DECISION j = i;
(TRUE):	(TRUE):
TASK i := inv(i);	TASK i := inv(i);
NEXTSTATE wait_put;	→ NEXTSTATE **WAIT_AM**;
ELSE:	ELSE:
...	...

Fig. 2. Process Mutant Operator Example: Destination State Exchanged

5.2 Interface Mutation

At program level, Delamaro et al. [10] defined Interface Mutation to test the interactions between the units compounding a software. Based on this concept, Souza et al. [29] defined interface mutant operators to Estelle applying the Interface Mutation to the specification level. In the same way, we propose an interface mutant operators set modelling communication errors among processes of an SDL specification, considering all the possible signals exchanges. Following these previous works, we divide the interface mutant operators in two groups: Group I, that explores the points where a process is called, i.e., at the *output* command; and Group II, that explores the process called, but the mutation is done in the *input* command and where computations are executed with the received signals.

– Example Group I: Output Missing
  This operator models output errors by excluding each output defined in the state transitions.
  One of the mutants generated when applying this operator to the Sender process of the Alternating-Bit protocol is presented in Figure 3. To generate this mutant, the operator is applied to the output of the transition of the `wait_put` state, excluding the "OUTPUT dm (m , i);" command.
  When this mutant is executed with ts the final states activated are $sa = <$ *([wait_am], [wait_dm, wait_am], [wait_dm])* $>$. The `wait_am` state is different from the expected one, the `wait_put` state. As a result, the mutant is distinguished from the original specification and so is considered dead. In this case, the test sequence $ts_i = <$*put(m), dm(m,0), dm(m,0), get(m)* $\wedge$ *am(0), medium_error*$>$ also distinguishes this mutant because the second signal `dm(m,0)` is not generated and the final states are $sa = <$ *([wait_am], [wait_dm, wait_am], [wait_dm])* $>$.

Original Specification	**Mutant Specification**
STATE wait_put;	STATE wait_put;
INPUT put (m);	INPUT put (m);
OUTPUT dm (m , i);	→
NEXTSTATE wait_am;	NEXTSTATE wait_am;
ENDSTATE;	ENDSTATE;

Fig. 3. Interface Mutant Operator Example: Output Missing

5.3 Structure Mutation

Structure Mutation explores errors in the architecture of the SDL specification that represents the hierarchical composition of the software components, their interaction and the data exchanged by them. The data flow is expressed by input

and output signals and the local variables of the processes. It is worth to note that some interface aspects are presented by the software structure, then we can also consider interface errors in the structure mutation context.

The structure mutant operators set models errors in the definitions of signal routes and channels, in the connections between them and in the list of signals declared.

– Example: Signal routes/Channels Missing
  This operator models errors related to the signal routes or channels definitions. It excludes each signal route and channel that links processes and blocks of the SDL specification.
  Figure 4 illustrates the definition of one signal route and one of the mutants generated when applying this operator to it. This mutant does not have the signal route that links the environment to the Sender process.
  When executing this mutant with *ts* the final states activated are *sa* = < *([wait_am], [wait_dm, wait_am], [wait_dm])* >. The `wait_am` state of the Sender process is different from the expected one which was the `wait_put` state. As a result, the mutant is distinguished from the original specification and so is considered dead. The test sequence ts_i = <*put(m), dm(m,0), dm(m,0), get(m)* $\wedge$ *am(0), medium_error*> also distinguishes this mutant because the last signal `medium_error` is not received by the Sender process and the final states are *sa* = < *([wait_am], [wait_dm, wait_am], [wait_dm])*>.

Original Specification	**Mutant Specification**
SIGNALROUTE sender_lower_interface FROM ENV TO sender_process WITH am , medium_error; FROM sender_process TO ENV WITH dm;	SIGNALROUTE sender_lower_interface → FROM sender_process TO ENV WITH dm;

Fig. 4. Structure Mutant Operator Example: Signal Routes/Channels Missing

5.4 Incremental Testing Strategies and Automation Aspects

Incremental Testing Strategies can be established to orient the application of the mutant operators to the SDL specifications. We can consider the three classes of operators and prioritize some aspects such as: behavior of the processes, communication among processes and structure of the specification. We can also select a subset of operators in each step of the strategy. The steps of one possible testing strategy are:

1. to validate the behavior of processes of the SDL specification, for each process
   (a) apply the Process Mutant Operators and determine an adequate test set;
2. to validate the communication among the processes, for each process

(a) apply the Interface Mutant Operators and determine an adequate test set;

3. to validate the structure of the SDL specification
   (a) for each signal route and channel definition
      – apply the Structure Mutant Operator (operators 1 to 4) and determine an adequate test set
   (b) for each connection between a channel and a signal route
      – apply the Structure Mutant Operators (operators 5 and 6) and determine an adequate test set

This testing strategy can be applied either top-down or bottom-up. In case an error is found during the application of this strategy, the specification should be corrected and the strategy applied again.

A testing tool to support the application of Mutation Testing is crucial since this activity can be error prone and unproductive if applied manually. Our work group has developed a family of tools to support Mutation Testing at the program and specification levels [22]. For testing C programs we have Proteum and Proteum/IM tools [9, 10]. For testing at the specification level, there are Proteum/FSM for the Finite State Machines based specifications [16]; Proteum/ST for Statecharts based specifications [18, 30]; and Proteum/PN for Petri Nets based specifications [28]. All these tools support the main functions related to the Mutation Testing: definition of a test case set, execution of the specification (or program), mutants generation, execution of the mutants, analysis of the mutants, computation of the mutation score and reports generation. When using these tools, the tester works in test sessions. In this way, the tester can start a test session, stop it at his/her convenience and resume the test session later from the point he/she has stopped. To allow this, the tools record the intermediate states of the test session. Information about mutants and test cases are maintained in a database. It is possible to select a subset of the mutant operators (constrained mutation) [23] or specify a percentage to be applied to generate the mutants. To analyze the mutants, the results obtained by their execution are compared to the result obtained by the original specification (or program) execution. Considering these features, we will develop a tool to support Mutation Testing to test SDL specifications. The preliminary results we present in this paper were obtained manually, but this was a very simple example and the development of a supporting tool is required. In fact we intend to add these functionalities into an existing tool named CAT_{SDL} [35], a coverage analysis tool that aids in testing specifications written in SDL. This tool supports control flow and data flow-based criteria.

Some problems existing in program testing also occur in specification testing, for example the oracle problem. The oracle problem remains in specification testing, either there is a formal mechanism to specify the expected behavior of the specification under test and then having an automated process to check the output or a human expertise is required. Other problem related to Mutation Testing is the high computational cost caused by the large number of mutants that can be generated. To overcome this problem at the program level, Offutt

et al. [23] and Barbosa et al. [4] proposed the determination of an essential operators set. In the same vein, we intend to investigate an essential operators set to SDL, based on our initial mutant operators set. At specification level, Simão and Maldonado [27] proposed as an alternative to overcome the computational cost of Mutation Testing applied to Petri Nets, the automatic generation of test sequences that in some cases identify equivalent mutants. We also intend to investigate this approach in the context of SDL specification testing.

6 Final Remarks

In this paper we proposed the use of Mutation Testing to test specifications written in SDL. We use this criterion as a mechanism for assessing the SDL specifications testing adequacy. Kovács et al. [20] use the test set generated by applying Mutation Testing to an SDL specification to test and validate the related implementation, i.e., it is applied in the conformance testing. Differently, we are interested in testing the specification itself. This is relevant so that the quality of the product can be guaranteed earlier in the development process. We were motivated by other works of our research group that have investigated Mutation Testing in the context of some formal techniques such as Finite State Machines, Statecharts, Petri Nets and Estelle. Although we are interested in specification testing, the test set generated based on the mutants can also be used to the conformance testing of implementations that claim to be conformed to the specification.

To propose a mutant operators set we took into account intrinsics features of SDL, the behavioral aspect of the processes, the communication among the processes and the structure of the SDL specification. These aspects were divided in three classes of mutant operators, that define a fault model to SDL: Process Mutation, Interface Mutation and Structure Mutation. Priority can be given to some aspects when the testing activity is conducted by a testing strategy. We also presented an incremental testing strategy for application of the Mutation Testing in this context.

The short term goals of our work on this subject is directed to three lines of research: improvement and refinement of the mutant operators (determining an essential operators set in the same line of Offutt et al. [23]), development of a tool to support Mutation Testing in the context of SDL and conduct empirical studies to compare Mutation Testing and Control and Data flow-based criteria.

Acknowledgements

The authors would like to thank the partial financial support from the brazilian funding agencies CNPq, CAPES and FAPESP, and from Telcordia Technologies (USA).

References

[1] Agrawal, H. (1989). Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Software Engineering Research Center/Purdue University.

[2] Ammann, P. and Black, P. (1999). A specification-based coverage metric to evaluate test sets. In *Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248. IEEE Computer Society.

[3] Ammann, P., Black, P., and Majurski, W. (1998). Using model checking to generate tests from specifications. In *Proceedings of 2^{nd} IEEE International Conference on Formal Engineering Methods*, pages 46–54, Brisbane, Australia. IEEE Computer Society.

[4] Barbosa, E. F., Maldonado, J. C., and Vincenzi, A. M. R. (2001). Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability Journal*, 11(2):113–136.

[5] Black, P. E., Okun, V., and Yesha, Y. (2000). Mutation operators for specifications. In *Proceedings of 15^{th} IEEE International Conference on Automated Software Engineering (ASE2000)*, pages 81–89.

[6] Bousquet, L. D., Ramangalahy, S., Simon, S., Viho, C., Belinfante, A., and Vries, R. G. (2000). Formal test automation: The conference protocol with TGV/TORX. In Ural, H., Probert, R. L., and v. Bochmann, G., editors, *IFIP 13^{th} International Conference on Testing of Communicating Systems(TestCom 2000)*. Kluwer Academic Publishers.

[7] Bromstrup, L. and Hogrefe, D. (1989). TESDL: Experience with generating test cases from SDL specifications. In *Proceedings of Fourth SDL Forum*, pages 267–279.

[8] Chow, T. S. (1978). Testing software design modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3):178–187.

[9] Delamaro, M. E. and Maldonado, J. C. (1996). Proteum: A tool for the assessment of test adequacy for C programs. In *Conference on Performability in Computing Systems*, pages 79–95, Brunswick, NJ.

[10] Delamaro, M. E., Maldonado, J. C., and Mathur, A. P. (2001). Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247.

[11] DeMillo, R. A. (1980). Mutation analysis as a tool for software quality assurance. In *Proceedings of COMPSAC80*, Chicago, IL.

[12] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41.

[13] DeMillo, R. A. and Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910.

[14] Fabbri, S. C. P. F., Maldonado, J. C., Delamaro, M. E., and Masiero, P. C. (1999a). Proteum/FSM: A tool to support Finite State Machine validation based on mutation testing. In *Proceedings of XIX SCCC - International Conference of the Chilean Computer Science Society*, pages 96–104, Talca, Chile.

[15] Fabbri, S. C. P. F., Maldonado, J. C., and Masiero, P. C. (1997). Mutation analysis in the context of reactive system specification and validation. In *5^{th} Annual International Conference on Software Quality Management*, pages 247–258, Bath, UK.

[16] Fabbri, S. C. P. F., Maldonado, J. C., Masiero, P. C., and Delamaro, M. E. (1994). Mutation analysis testing for Finite State Machines. In *Proceedings of ISSRE'94 - Fifth International Symposium on Software Reliability Engineering*, pages 220–229, Monterey, California, USA.

[17] Fabbri, S. C. P. F., Maldonado, J. C., Masiero, P. C., Delamaro, M. E., and Wong, E. (1995). Mutation testing applied to validate specifications based on Petri nets. In *Proceedings of FORTE'95 - 8^{th} International IFIP Conference on Formal Description Techniques for Distributed Systems and Communications Protocol*, pages 329–337, Montreal, Canada.

[18] Fabbri, S. C. P. F., Maldonado, J. C., Sugeta, T., and Masiero, P. C. (1999b). Mutation testing applied to validate specifications based on Statecharts. In *ISSRE — International Symposium on Software Reliability Systems*, pages 210–219, Boca Raton, Flórida, EUA.

[19] Ghosh, S. and Mathur, A. P. (2000). Interface mutation. In *Mutation 2000 - A Symposium on Mutation Testing for the New Century*, pages 112–123, San José, California.

[20] Kovács, G., Pap, Z., Viet, D. L., Wu-Hen-Chang, A., and Csopaki, G. (2003). Applying mutation analysis to sdl specifications. In Reed, R. and Reed, J., editors, *SDL 2003: System Design - 11^{th} SDL Forum*, volume 2708 of *Lecture Notes on Computer Science*, pages 269–284, Stuttgart, Germany.

[21] Luo, G., Das, A., and Bochmann, G. (1991). Software test selection based on SDL specification with save. In *Proceedings of 5^{th} SDL Forum*, pages 313–324, Glasgow. Elsevier.

[22] Maldonado, J. C., Delamaro, M. E., Fabbri, S. C. P. F., Simão, A. S., Sugeta, T., Vincenzi, A. M. R., and Masiero, P. C. (2000). Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation 2000 - A Symposium on Mutation Testing for the New Century*, pages 146–149, San José, California.

[23] Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., and Zapf, C. (1996). An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118.

[24] Offutt, A. J. and Pan, J. (1997). Automatically detecting equivalent mutants and infeasible paths. *The Journal of Software Testing, Verification, and Reliability*, 7(3):165–192.

[25] Petrenko, A. and Bochmann, G. (1996). On fault coverage of tests for Finite State specifications. Technical report, Département d'Informatique et de Recherche Opérationnelle, Université de Montreal.

[26] Probert, R. L. and Guo, F. (1991). Mutation testing of protocols: Principles and preliminary experimental results. In *Proceedings of the IFIP TC6 Third International Workshop on Protocol Teste Systems*, pages 57–76, North-Holland.

[27] Simão, A. S. and Maldonado, J. C. (2000). Mutation based test sequence generation for Petri nets. In *Proceedings of III Workshop of Formal Methods*, pages 68–79, João Pessoa, PB.

[28] Simão, A. S., Maldonado, J. C., and Fabbri, S. C. P. F. (2000). Proteum-RS/PN: A tool to support edition, simulation and validation of Petri nets based on mutation testing. In *Proceedings of XIV Brazilian Symposium of Software Engineering*, pages 227–242, João Pessoa, PB, Brazil.

[29] Souza, S. R. S., Maldonado, J. C., Fabbri, S. C. P. F., and Lopes de Souza, W. (2000). Mutation testing applied to Estelle specifications. *Quality Software Journal*, 8(4):285–301. (Also published in the 33^{rd} Hawaii Internacional Conference on System Sciences - 2000).

[30] Sugeta, T., Maldonado, J. C., Masiero, P. C., and Fabbri, S. C. P. F. (2001). Proteum-RS/ST – A Tool to Support Statecharts Validation Based on Mutation Testing. In *4^{th} Workshop Iberoamericano de Engenharia de Requisitos e Ambientes de Software - IDEAS'2001*, pages 370–384, Santo Domingo, Costa Rica.

[31] Tanenbaum, A. S. (1996). *Computer Networks*. Prentice Hall, 3 edition.

[32] Ural, H., Saleh, K., and Williams, A. (2000). Test generation based on control and data dependencies within system specifications in SDL. *Computer Communications*, 23(7):609–627.

[33] Weyuker, E. J., Goradia, T., and Singh, A. (1994). Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363.

[34] Wong, W. E., Maldonado, J. C., and Mathur, A. P. (1994). Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness. In *First IFIP/SQI International Conference on Software Quality and Productivity – Theory, Practice, Education and Training*, Hong Kong.

[35] Wong, W. E., Sugeta, T., Li, J. J., and Maldonado, J. C. (2003). Coverage testing software architectural design in SDL. *Computer Networks*, 42(3):359–374.