Mutation Testing Applied to Estelle Specifications

Article *in* Software Quality Journal · December 1999

DOI: 10.1023/A:1008978021407 · Source: DBLP

CITATIONS
38

READS
60

4 authors, including:

Simone Do Rocio Senger de Souza
University of São Paulo
111 PUBLICATIONS **819** CITATIONS
SEE PROFILE

José Carlos Maldonado
University of São Paulo
339 PUBLICATIONS **4,435** CITATIONS
SEE PROFILE

Wanderley Lopes de Souza
Universidade Federal de São Carlos
83 PUBLICATIONS **367** CITATIONS
SEE PROFILE

Mutation Testing Applied to Estelle Specifications

Simone do Rocio Senger de Souza*
Universidade Estadual de Ponta Grossa - UEPG
Departamento de Informática
rocio@icmc.sc.usp.br

José Carlos Maldonado
Universidade de São Paulo – ICMC/USP
Departamento de Computação e Estatística
jcmaldon@icmc.sc.usp.br

Sandra Camargo Pinto Ferraz Fabbri
Universidade Federal de São Carlos-UFSCar
Departamento de Computação
sfabbri@dc.ufscar.br

Wanderley Lopes de Souza
Universidade Federal de São Carlos-UFSCar
Departamento de Computação
desouza@dc.ufscar.br

Abstract

Many researchers have pursued the establishment of a low-cost, effective testing and validation strategy at the program level as well as at the specification level. Mutation Testing is an error-based approach, originally introduced for program testing, that provides testers a systematic way to evaluate how good a given test set is. Some studies have also investigated its use to generate test sets. In this article, the application of Mutation Testing for validating Estelle specifications is proposed. A mutation operator set for Estelle – one of the crucial points for effectively applying Mutation Testing – is defined, addressing: the validation of the behavior of the modules, the communication among modules and the architecture of the specification. In this scope, these operators can be taken as a fault model. Considering this context, a strategy for validating Estelle-based specification is proposed and exemplified using the Alternating-bit protocol.

1. Introduction

Software testing, which has as objective the identification of not-yet-discovered errors, is one of the most important activities to guarantee the quality and the reliability of the software under development. The success of the testing and validation activities depends on the quality of a test set. Criteria that allow selecting a subset of the input domain preserving the probability of revealing the existent errors in the program or specification are necessary, since the exhaustive test is, in general, impracticable. These criteria systematize the testing activity and may also constitute a coverage measure [5, 10, 19, 22].

Some empirical studies have provided evidences that Mutation based criteria are among the most promising ones in terms of fault detection [20, 27, 28]. Basically, this approach consists of generating mutants of the program, based on a mutation operator set. This set is intended to model 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. This approach also provides mechanisms to evaluate the quality of a test set and/or to generate test sets [11].

For safety critical applications, errors can produce disastrous consequences, therefore the use of formal techniques should be seriously considered, if not mandatory in this context. Some examples of such systems are air traffic control, bank control, patients monitoring and communication protocols. Formal Description Techniques (FDTs) provides means to elaborate consistent, sound, concise, not ambiguous software specifications [4, 18].

Formal Description Techniques provide facilities for specification testing, verification and validation activities. Moreover, the earlier the errors are discovered in the software development process less difficult is the task of removing them. These aspects motivate the definition of criteria, methods and strategies for specification testing aiming to identify errors in the initial phases of the development process. Thus, starting from a specification S, to verify if S satisfies the user requirements, a test set should be selected and/or evaluated. Probert and Guo [21] and Fabbri [14] have investigated the adequacy of applying mutation testing to validate formal specifications: Estelle and Statecharts, respectively.

Estelle is a FDT that describes the system through a hierarchy of Extended Finite State Machines that communicate through bi-directional channels. Estelle has been developed by ISO (International Organization for

* PhD student at Instituto de Física de São Carlos - IFSC/USP

Standardization) for distributed system, service and protocol specification [17].

Probert and Guo [21] introduced the approach E-MPT (*Estelle-directed Mutation-based Protocol Testing)* that applies Mutation Testing to validate the behavior of Estelle specifications. In fact, their approach addresses the validation of the Extended Finite State Machines defined by the specification. Two mutation types are defined: major mutation – which tests the basic structures of Estelle – and minor mutation – which tests the correctness of the operations of the transitions. The generation of the mutant specifications is based on the Finite Complete Set of Alternatives, which possesses, for each element that can suffer a mutation (variables, constants and mathematical operators) its syntactically correct alternatives based on the specification under testing.

Fabbri et al defined Mutation Testing to validate specifications based on Petri Nets [13], Finite State Machines (FSM) [12] and Statecharts [14]. For each specification technique a mutation operator set has been defined inspired in the error classification suggested by Chow [6]. For the Statecharts technique, abstraction strategies were proposed exploring its features and allowing to select its basic components (EFSM-Extended Finite State Machines), in different hierarchy levels.

Other relevant work that this paper is based on is the *Interface Mutation* criterion defined by Delamaro [8,9]. Interface Mutation is defined for Integration Testing, exploring interface errors related to the connections among program units, in a pairwise approach.

The objective of this paper is to provide mechanisms to apply Mutation Testing to validate Estelle specifications. We elaborate on Probert and Guo's work taking into account the basic ideas and mutant operators explore in Fabbri et al's work. Mutation Testing is also investigated to validate the communication among the modules and the structure of the specification, not only to the behavioral aspect of the Estelle specification. As the behavior of the modules is described through EFSM, the mutation operators defined by Fabbri are revisited considering the intrinsic features of Estelle. The mutation operators to validate the communications among modules are defined exploring the same ideas of the Interface Mutation.

This paper is organized as follows: Estelle basic concepts are presented in Section 2 and Section 3 contains a synthesis of Mutation Testing. In Section 4 the mutation operators for Estelle are defined as well as a mutation-based validation strategy for Estelle specification. The Alternating-bit Protocol [17] is used to illustrate the ideas introduced herein and preliminary results are presented in Section 5. Final remarks are discussed in Section 6.

2. Estelle: Basic Concepts

A system specified in Estelle is structured in a hierarchy of communicating modules. Extended Finite State Machines (EFSM) model the behavior of each module. The exchanges of messages (interactions) carry out the communication among the modules. These messages are stored in an infinite FIFO queue and processed according to the conditions, priorities and delays associated to the transitions of the related EFSM.

In Estelle, there are notions of parent and child modules – a module and one of its immediate sub-modules; of siblings – children of the same parent module; and of ancestor and descendants – the transitive closure of the parent and child relations, respectively [4]. A parent may share those variables that the child chooses to *export*. Estelle semantic guarantees that there are no synchronization problems with shared variables (*parent/children priority principle*).

There are different communication and synchronization possibilities among a module and its descendants, among siblings, or between two modules with no descending relation. The synchronization and communication are determinate by the module attribute: *systemprocess*, *systemactivity*, *process* and *activity*.

System modules (*systemprocess* and *systemactivity)* are subsystems executed asynchronously, in parallel. Inside a *systemprocess* module, synchronous parallelism or sequential execution can occur depending on the attribute of its child modules: *process* or *activity*. For instance, the children of a *process* module can be *process* or *activity* and are executed synchronously, in parallel; in other words, a transition of each module is selected in each step and these transitions are executed in parallel. On the other hand, the children of a module *activity* can only be *activity* and are executed sequentially: only one ready-to-fire transition is randomly selected for execution in each step. Considering a *systemactivity* module, its child modules can only receive the attribute *activity* so that only sequential executions occur [4, 18].

In an Estelle specification, several instances of modules and connections may be defined. These instances can be created static or dynamically. The static structure once established cannot be modified at execution time. The dynamic structure is created at runtime: module instantiation and destruction are associated to the firing of the transitions.

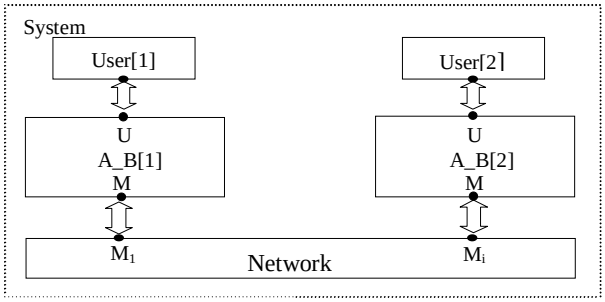

Figure 1. Alternating-bit protocol [17].

Figure 1 presents a structure for Alternating-bit protocol specification composed of three modules: *User*, *A_B* and *Network* [17]. The modules *User* and *A_B* possess two instances each: *User[1], User[2], A_B[1]* and *A_B[2]*. Two instances of the channel U connect the modules *User[1]* with *A_B[1]* and *User[2]* with *A_B[2]*. Two instances of the channel M connect the modules *A_B[1]* and *A_B[2]* with the module *Network*. This protocol is used in Section 5 to illustrate the application of the Mutation Testing approach to Estelle.

3. Mutation Testing

Mutation Testing is used to increase the confidence that a program P (or a specification S) is correct by producing, through small syntactic changes, a set of mutant elements that are similar to the product under test. These changes are based on an operator set called *mutation operators*. To each operator it is associated an error type or an error class that we want to reveal in the program. A test case set that is capable of causing behavioral differences between P (or S) and each one of its mutants is then created.

The definition of the mutation operators is a key factor for the success of Mutation Testing. At the program level, 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 [10] 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 (or specification S) 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 S, it is said to be dead, otherwise, it is said to be alive. In this case, either there is no test case in T that is capable to distinguish M from P (or S) or M and P (or S) are equivalent, in the sense that 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 able to kill all non-equivalent mutants; in this case T is considered adequate to test P (or S).

DeMillo notes that Mutation Testing provides an objective measure for the confidence level of the test case set adequacy [11]. 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 specification under testing.

It is recognized that the major obstacle to the use of Mutation Testing is its computational cost due to the high number of mutants and to the need to analyze them aiming to determine the equivalent ones. Since the equivalence question is, in general, undecidable, the equivalent mutants are obtained interactively as an entry from the tester. Wong et al provide evidences that generating only a small percentage of the mutants may be a useful heuristic for evaluating and constructing test sets in practice [27]. In constrained mutation the mutant generation is constrained by fixing the mutant types to be generated and in randomly selected mutation, only a small percentage of each mutant type is examined. Offut et al [20] and Barbosa et al [3] 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 mutation operators.

4. Mutation Testing Applied to Estelle

The Mutation Testing criterion was originally used for program test. In this paper, it is applied to the validation of Estelle specifications. With this purpose in mind, the basic hypotheses of the Mutation Testing criterion at the specification level were revisited [12,13,14,21]. In the design of the mutation operators, the designer competent hypothesis and the coupling effect that, in fact, must be validated at this level, were considered. So, 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 or Z differs from S at least on a test point.

Experimental results obtained by Fabbri et al applying Mutation Testing, in the context of protocol specifications described in terms of FSMs, indicate that this criterion can contribute to the improvement of the testing activity [12]. In fact, the coverage provided by this criterion is relevant and may complement the methods of test sequence generation for FSMs. Ural describes the main methods of test sequence generation for FSMs [25]. These works motivate the study of the adequacy to apply Mutation Testing in the context of Estelle since the usual

approaches for FSMs and EFSMs validation have also been investigated for Estelle [2, 15, 16].

As pointed out before, the mutation operator set is a crucial point for effectively applying the mutation concept. For Estelle, the mutation operator set used was inspired in the control structure sequencing error classes defined by Chow [6], in the mutation operators for C language [1, 7, 9] and in the mutation operators for boolean expressions, defined by Weyuker [26]. Fabbri et al have used these ideas in the context of Statecharts [14].

The Mutation Testing for Estelle proposed herein is divided in three categories: Module Mutation, Interface Mutation and Structure Mutation. These categories are described below and, for each one, the definition of one of the mutation operators is presented informally. A synthesis of the mutation operators of each category is given in Table 1.

- **Module Mutation**

Module mutation models errors related to the behavior of the modules. As the behavior of the modules in Estelle is expressed by EFSM, the mutation operators defined by Fabbri for EFSMs [14] are included in the mutation operator set defined for modules. Other mutation operators are defined to cover intrinsic aspects of Estelle :

- The existence of queues that store the interactions received by the module;
- The clause *delay* that determines a delay for the firing of the transitions;
- The clause *priority* that defines a priority for the transitions.

Example - *Origin state exchanged operator*

This operator models transfer errors, mutating the origin state of the transitions. The origin state of a transition *t* is exchanged by other states defined from the state-definition-part of the module containing *t.*

- **Interface Mutation**

The interface mutation has as main objective to validate the interactions among the units that compose the software. This idea has been proposed to apply Mutation Testing for integration testing at program level [8,9]. This fact, translated to Estelle, means to validate the communications among pairs of connected modules.

Communications in Estelle happen in two ways: 1) exchange of messages and 2) shared variables. The exchange of messages occurs between two modules connected by a bi-directional channel, as illustrated in Figure 2. The module *A* sends a message (a primitive of communication and its parameters) to the module *B*. The received message is stored in a queue until it is processed after a transition in *B*, according to the semantics of the language [4, 18]. In the same way, the module *B* can also send messages to the module *A*.

The messages produced are described by the command *output ip.primitive(parameters)* associated to the transitions, where *ip* represents the interaction point by which the message will be sent.

The second communication type (shared variables) only happens between a parent module and a child module. The declaration of a variable of the type export in a child module means that the parent module has access to its value. The simultaneous access of variables of type export is not possible because of the *parent/children priority principle* (Section 2). The interface mutation intends to model the following types of errors:

- Errors in the data flow among the components;
- Errors in the communication primitives sent or received by the components and/or in shared variables;
- Errors in the computation of the primitives received. and/or in shared variables

Following, in the same vein of the Delamaro et al's work [8,9], the interface mutation operators for Estelle are divided in two groups (see Table 1):

1. Mutations in the calling point: the operators of this set are applied where a call to a module occurs, i.e., in the command *output*. The operators are mainly applied in the primitive sent and in its parameters. The *interface variables* are considered the ones passed as parameters and the shared variables.

2. Mutations in the called module: the operators of this group apply the mutations in the called module. The operators of this set are only applied in the commands that execute computations with the received data.

Example Group I: *Output exchanged operator*

This operator model errors related with data sent to a wrong module. This operator mutates the output of the transitions, exchanging the output of the each transition by outputs of the others transitions of the module.

Example - Group II: *Non-interface variables exchanged operator*

This class of operators mutates variables or constants that may influence interface values. This operator is applied in each non-interface variable that occurs in expressions involving interface variables, exchanging non-interface variables by other non-interface variables of compatible type, defined for the module or defined for the ancestral modules.

Figure 2. Two modules connected.

- **Structure Mutation**

When a system is specified in Estelle, it is possible to describe its structure (or architecture), either static or dynamic, by defining the hierarchy among the modules and their connections.

The structure mutation testing intends to model architectural errors in Estelle specifications. Tracz defines some essential properties of software architecture: *partitioning of the software components*; *flow of data and*

control; *critical timing and throughput attributes*; *interconnection; and allocation of software to hardware* [24]. These properties served as the base to identify errors related to the structure:

- Incorrect partition of the system components;
- Components not static or dynamically initialized;
- Connections among components missing;
- Error in the finalization of a connection; and
- Error in the control flow among the components, that is, in the synchronization and parallelism of the components.

Based on these errors, the structure mutation operators are proposed. (See Table1)

Example : *Connection missing Operator*

This operator models errors related with missing connections among components. This operator removes each connection *connect* or *attach* type between modules. The difference of these types is that *attach* connections can only occur between a parent module and a child module, so that all the received messages by the parent module are directly passed to its child modules.

- **Strategies**

The division of the mutation operators in testing categories according to Estelle features provides one mechanisms for the establishment of Incremental Testing Strategies, either top-down or bottom-up, exploring or prioritizing specifics aspects such as: behavior of the modules, communication, parallelism and structuring (static or dynamics). Moreover, one can decide which mutation operators to use in each step of the strategy. A possible strategy is as follows:

i) for each module of Estelle, apply the Module Mutation operators and determine an adequate test set;

ii) determine all possible connections among modules;

- for each connection, apply the Interface Mutation operators and determine an adequate test set;

iii) apply the Structure Mutation operators (operators number 4 to 7 of Table 1) and select an adequate test set to validate the parallelism of Estelle;

iv) apply the Structure Mutation operators (operators number 1 to 3 of Table 1) and determine an adequate test set to validate both the static structure and the dynamic structure.

In case an error is found during the application of this strategy, the specification should be corrected and the strategy applied again. The high number of mutants that can be generated may compromise the practical application of the Mutation Testing. In this sense, we intend to investigate the definition of an essential set of mutation operators for each testing category, in the same vein of Offut et al [20] and Barbosa [3] work as well as *alternative mutation* proposed by Wong et al [27].

In the next section an example illustrating the application of these ideas is presented. As this procedure is not yet automated the results were obtained manually and, for that, it was chosen a simple example.

Table 1. Estelle: mutation operators.

Module Mutation Operators
1. Initial state exchanged
2. Origin state exchanged
3. Tail state exchanged
4. State missing
5. Transition missing
6. Condition missing
7. Event exchanged
8. Event missing
9. Negation of the condition
10. Mathematics operators exchanged
11. Negation logical
12. Variable by variable replacement
13. Variable by constant replacement
14. Constant by constant replacement
15. Increment/decrement variables/constants
16. Unary operator inclusion in variable
17. *Begin-end* blocks exchanged
18. Coverage of code (*begin-end* block)
19. *Priority* clause missing
20. *Delay* clause missing
21. Queue policies exchanged
Interface Mutation Operators
Group I: Calling Point
1. Parameters exchanged
2. Increment and decrement of parameters
3. Order of the parameters exchanged
4. Unary operator inclusion in parameters
5. Output exchanged
6. Output missing
Group II: Called Module
7. Interface variables exchanged
8. Non-interface variable exchanged
9. Interface variable increment/decrement
10. Unary operator inclusion in variable
Structure Mutation Operators
1. Connection missing
2. Disconnect added
3. Disconnect missing
4. Sync. parallelism by sequential
5. Sync. by async. parallelism
6. Sequential execution by sync. parallelism
7. Sequential execution by async. parallelism

5. Example: Alternating-bit Protocol

The example of the Alternating-bit protocol [17] is well known and is used to illustrate the application of the Mutation Testing concepts to Estelle. The structure of the system presented in Figure 1, is composed of a main module, called *system*, divided in three child modules. Figure 3 presents the EFSMs of the each module.

- ***User:*** describes the connected users. These users can send and receive messages.
- ***A_B:*** describes an Alternating-bit protocol that aims to provide reliable communication over a non-reliable

network service. This protocol uses a one-bit sequence number (which alternates between 0 and 1) in each message or acknowledgement to determine when messages must be retransmitted [17].

- ***Network:*** description of the communication medium that receives packages from users and sends them to the addressed users.

To illustrate the ideas presented in this paper, the test sequence is generated taking into account only the functionality of the protocol. At first we took a typical transition sequence *ts=<sendit, send, req1, getdatae, req2, goodack>* that corresponds to:

User 1 sends a message to User 2 (sendit transition). A_B protocol of User 1 packs the message and sends the first data sequence to the Network (send transition). The Network sends to User 2 (req1 transition). A_B protocol of the User 2 receives the data from Network, stores and sends acknowledgement to User 1 (getdatae transition). The Network sends the acknowledgement received from User 2 to User 1 (req2 transition). The A_B protocol of the User 1 receives the acknowledgement (goodack transition).

These transitions are highlighted in Figure 3. When it is executed the following output sequence is obtained: *os=<send_request, data_request, data_response, data_request, data_response>*, that corresponds to the primitives sent when each transition is fired.

For each category of the mutation testing defined in Section 6 a mutation operator was chosen to provide an overview of the application of the Mutation Testing. The test sequence *ts* above is executed with the mutants generated by the chosen operators and the results are presented.

5.1. Module Mutation Operators

The *origin state exchanged* operator is used to exemplify the Module Mutation operators. It is applied in the module *A_B* (Figure 3) in the transition *send*. The origin state of this transition is *estab*. The mutants are generated selecting the other states defined for the module *A_B*; in this case, the states *ackwait* and *either* (*either* is a *stateset* composed of the *estab* and *ackwait*) are selected. Figure 4 presents the description of this transition and one of the mutants generated. For the other transitions the mutants are generated in the same way.

The mutant specification (Figure 4) when executed with the *ts* produces the output *send_request*. The initial state of module *A_B* is *estab* and the *send*-transition cannot be executed as its origin state is *ackwait*. The primitive *send_request* stays in the queue of the module *A_B* waiting for the occurrence of a transition that processes it. Therefore, the mutant is distinguished from the original specification and is considered dead.

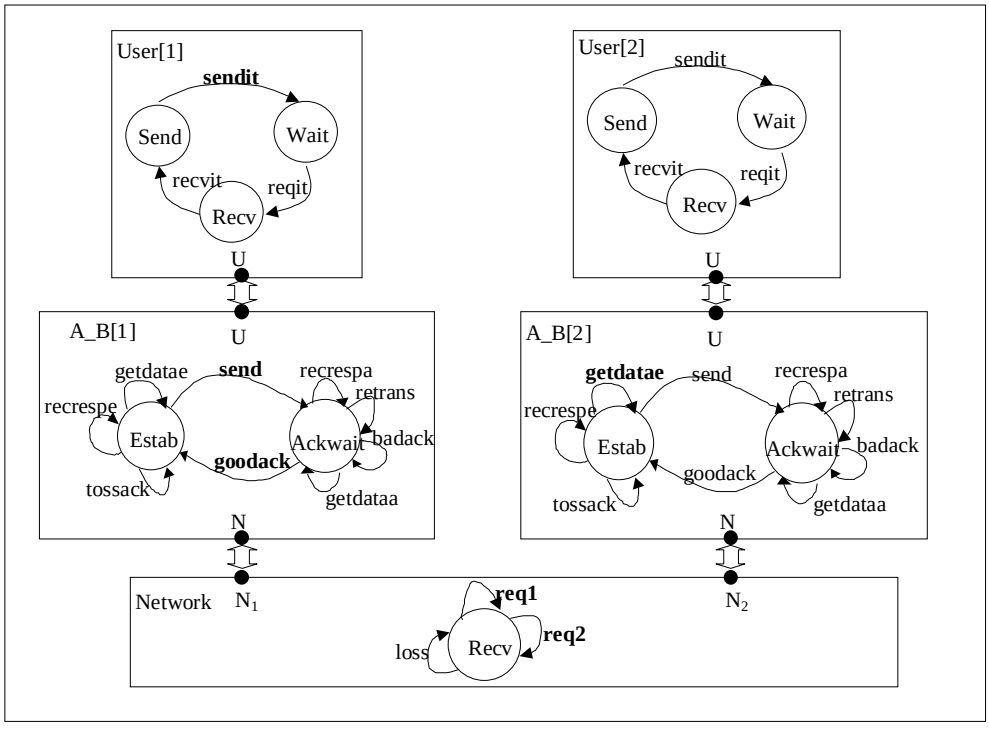

Figure 3. Finite state machines of the modules.

Transition 1

```
trans {1}
 from ESTAB
 to ACKWAIT
 when U.Send_Request
 name send
 begin
   copy(P.Msgdata,Udata);
   P.Msgseq := Sendseq;
   store(Sendbuffer, P);
   format_Data(P,B);
   output N.Data_Request(B);
end;
```

Mutant of Transition 1

```
trans {1}
 ➔ from ACKWAIT
 to ACKWAIT
 when U.Send_Request
 name send
 begin
     copy(P.Msgdata,Udata);
     P.Msgseq := Sendseq;
     store(Sendbuffer, P);
     format_Data(P,B);
     output N.Data_Request(B);
end;
```

Figure 4. Example: module mutation operator.

5.2. Interface Mutation Operators

For the Interface Mutation operators were chosen one mutation operator from each group: *output exchanged operator* and *non-interface variables exchanged operator*

Before generating the mutants, it is necessary to identify the interface variables and the non-interface variables for each connection. Consider that *f* represents the calling module and *g* is the called module. The interface variable set consists of the variables passed to *g* by input parameters and the export variables. The non-interface variable set consists of the local variables of *g*, the export variables not used in *g* and the constants of *g*.

Considering the connection between module *User* and module *A_B* associated to the transitions illustrated in the Figure 5 and 6, the following sets are defined:

Interface variable set = *{udata}*

Non-interface variable set = *{P.msgdata, Q.msgdata, B.data, P.msgseq, Q.msgseq, B.seq, sendseq, recvseq, sendbuffer, recvbuffer, sendseq, P, Q, B}*

In Figure 5 one of the mutants generated by the mutation operator *output exchanged* applied to the module *User (calling module)* is presented. It is generated exchanging the output of the transition sendit; the output of the mutant transition is *U.receive_request.* When this mutant is executed with the *ts* it produces the output *receive_request.* The transition *send* of the module *A_B* can not execute because the primitive *send_request* is not in the queue U. Therefore, this mutant specification is distinguished from the original one.

In Figure 6 one mutant generated for the mutation operator *non-interface variables exchanged* is presented. This operator is applied in the module *A_B* (*called module).* The mutant illustrates the exchanging of *P.Msgdata* variable by *Q.Msgdata* variable. This mutant when executed with *ts* produces the same output produced by the original specification. This means that the sequence is not good enough for distinguishing this mutant. In order to distinguish this mutant it is necessary to design one test sequence that would make the transition *recrespe* to send data to *User 2* and the transition *recvit* to receive the data in *User 2.*

Calling Module

```
Module User
...
trans {1}
 from SEND
 to WAIT
 delay(30);
 name SENDIT
 begin
  Udata.size := 5;
  Udata.info := 'Hello   ';
  output U.Send_Request(Udata);
 end;
```

Mutant

```
Module User
...
trans {1}
 from SEND
 to WAIT
 delay(30);
 name SENDIT
 begin
  Udata.size := 5;
  Udata.info := 'Hello   ';
➔output U.Receive_Request;
end;
```

Figure 5.Example: output exchanged operator.

5.3. Structure Mutation Operators

The *connection-missing* operator is used to exemplify the Structure Mutation operators. This operator removes all the occurrences of *connect* and *attach* statements, one at a time. Figure 7 presents part of the initialization of the modules and connections for the original and for the mutant specifications. In the mutant the connection between module *User* and module *A_B* is missing.

This mutant when executed with the *ts* produces the output *send_request.* The other transitions cannot be executed because, when the module *User* tries to send the primitive *send_request* to module *A_B*, there is no connection between these modules and a deadlock occurs.

Table 2 presents the total number of mutants generated by all the operators: module mutation, interface mutation and structure mutation. These results were obtained

manually. For the module mutation was considered only the module *A_B*. In the same way, for the interface mutation operators were considered the connections between the others modules with module *A_B*.

Some operators have not generated mutants for the Alternating-bit protocol specification. For instance, considering the interface mutation operators, most of the communication primitives do not have parameters so that the operators from 1 to 4 of Table 2 do not generate mutants. The structure mutation operators related to dynamic structure did not generate mutants as well, since this kind of structure does not occur in this case.

The application of the Mutation Testing without the support of a testing tool is impracticable. The definition of a testing tool to support the ideas presented in this paper has been investigated. Mutation Testing tools that has been developed in our work group provides an essential base for the definition of a tool for the validation of Estelle specification. These tools are *Proteum* and *Proteum/IM (Program Testing Using Mutants) at the program level* for the validation of C programs [7, 8]. At the specification level, *Proteum/FSM* for the validation of Finite State Machines based specifications [12]; *Proteum/RS* for the validation of Statecharts based specifications [14]; and *Proteum/PN* for the validation Petri Nets based specifications [23].

The main functions of these tools correspond to the activities of the Mutation Testing: test case definition; specification execution; mutant generation; mutant execution; mutant analysis; mutation score calculus and report generation. A database maintains the information about both the test cases and the mutants with their status. These tools have been developed to allow the user to work in test sessions, enabling him to start a specification test, stopping the activity at his convenience, and restarting the test from the point that he has stopped. For this purpose, the intermediate states of the test session are recorded and can be resumed later. Facilities for applying either constrained mutation or randomly mutation are also provided; the tester can select a subset of the mutation operator set and also specify a percentage that is to be applied when generating the mutants.

In synthesis, once a set of test sequences (test case set) has been designed, it can be applied to the already generated mutants. The results obtained by executing the mutants are compared to the results obtained executing the original specification. The status of a test session can be checked at any moment and a summary of the most relevant information obtained, such as: the number of mutants generated, the number of equivalent mutants, the number of alive mutants, the number of executed mutants and the mutation score. Based on this information, the tester can proceed the analysis of the mutants alive to determine if they are equivalent or if test cases that would kill them can be determined. In the first case, the tester would mark them as equivalent. In the second case, the tester would add new test cases, improving the test case set at hand and the mutation score; in other words, improving the testing activity conducted up to that point.

Called Module

```
Module A_B
...
trans {1}
   from ESTAB
   to ACKWAIT
         when U.Send_Request
         name send
    begin
          copy(P.Msgdata,Udata);
          P.Msgseq := Sendseq;
          store(Sendbuffer, P);
          format_Data(P,B);
          output N.Data_Request(B);
         end;
```

Mutant

```
Module A_B
...
trans {1}
   from ESTAB
   to ACKWAIT
   when U.Send_Request
   name send
   begin
➔   copy(Q.Msgdata,Udata);
    P.Msgseq := Sendseq;
    store(Sendbuffer, P);
    format_Data(P,B);
    output N.Data_request(B);
   end;
```

Figure 6. Example: non-interface variable exchanged operator.

Original Specification

```
...
     init Alternatingbit[Cep] with
                    Alternatingbitbody(Cep);
     connect User[Cep].U to
                    Alternatingbit[Cep].U;
     connect Alternatingbit[Cep].N to
                    Network.N[Cep];
   end;
...
```

Mutant

```
...
     init Alternatingbit[Cep] with
                   Alternatingbitbody(Cep);
 ➔   ;

     connect Alternatingbit[Cep].N to
                    Network.N[Cep];
   end;
...
```

Figure 7.Example: structure mutation operator.

Table 2. Total of the mutants.

Mutation Operators	# mutants
Module Mutation	
1. Initial state exchanged	01
2. Origin state exchanged	18
3. Tail state exchanged	18
4. State missing	0
5. Transition missing	09
6. Condition missing	07
7. Event exchanged	0
8. Event missing	0
9. Negation of the condition	07
10. Mathematics operators exchanged	26
11. Negation logical	0
12. Variable by variable replacement	92
13. Variable by constant replacement	0
14. Constant by constant replacement	02
15. Increment/decrement variables/constants	20
16. Unary operator inclusion in variable	10
17. *Begin-end* blocks exchanged	22
18. Coverage of code (*begin-end* block)	13
19. *Priority* clause missing	0
20. *Delay* clause missing	01
21. Queue policies exchanged	02
TOTAL	**248**
Interface Mutation	
1. Parameters exchanged	0
2. Increment and decrement of parameters	0
3. Order of the parameters exchanged	0
4. Unary operator inclusion in parameters	0
5. Output exchanged	18
6. Output missing	18
7. Interface variables exchanged	498
8. Non-interface variable exchanged	371
9.Interface variable increment/decrement	0
10.Unary operator inclusion in variable	32
TOTAL	**937**
Structure Mutation	
1. Connection missing	02
2. Disconnect added	0
3. Disconnect missing	0
4. Sync.parallelism by sequential	0
5. Sync. by async. parallelism	0
6. Sequential by sync. parallelism	01
7. Sequential by async. parallelism	02
TOTAL	**05**

6. Final Remarks

The development of a low-cost and effective testing strategy constitute the motivation of several researches, since testing is one of the most expensive and time consuming of the software development activities. In this paper the use of Mutation Testing for validation Estelle based specifications was proposed as a mechanism for testing adequacy assessment in this context. The main contribution of this paper is the proposition of a mutation operator set for Estelle, a key point for the successful application of Mutation Testing, providing mechanisms for the establishment of a mutation-based validation strategy for Estelle.

Previous works, as the work of Fabbri [12, 14] and Probert and Guo [21], gave the basic motivation and guidelines for the definition of the Estelle mutation operator set proposed in this work. The mutation operator set defined takes in consideration the intrinsic characteristics of Estelle, synchronous and asynchronous parallelism, as asynchronous communication, and dynamic structures among others. They are divided in three categories: Module Mutation, Interface Mutation and Structure Mutation. These operators define a fault model for Estelle. A strategy for application of the Mutation Testing was also illustrated, making feasible to conduct the validation activity giving priority to specific types of errors. These ideas also make feasible to establish an incremental testing strategy in the context of Estelle.

The application of the mutation operators was exemplified with the Alternating-bit protocol [17]. The results obtained manually indicated the need for a testing tool to support the application of Mutation Testing for validation of Estelle specification.

The evolution of our work on this subject is directed to three lines of research: refinement of the mutation operators, development of a tool to apply Mutation Testing in the context of Estelle; and conduct empirical studies.

Acknowledgements

The authors would like to thank the Brazilian funding agencies CAPES, FAPESP and CNPq for their support to the research activities carried out by the Software Engineering Group at ICMC/USP, São Carlos, Brazil.

References

[1]. Agrawal, H. et al. "Design of Mutant Operators for the C Programming Language", *Technical Report SERC-TR-41-P*, Purdue University, March, 1989.

[2] Amer, P.D., Sethi, A.S., Fecko, M., Formal Design and Testing of Army Communication Protocols Based on Estelle, *proc. 1st ARL/ATIRP Conference*, College Park, pp.107-114, 1997.

[3] Barbosa, E. F., Vincenzi, A. M. R., Maldonado, J.C., A Contribution for the Determination of a Essential Mutation Operator Set for C Programs, *in proc. XII SBES'98 – Brazilian Software Engineering Symposium*, Maringá, PR, October, 1998. (In Portuguese)

[4] Budkowski, S., Dembinski, P., An Introduction to Estelle: a specification language for distributed systems, *Computer Network and ISDN Systems*, v14(1), pp.3-23, 1987.

[5] Bochmann, G.v., Petrenko, A., Protocol Testing: Review of Methods and Relevance for Software Testing, *in proc. ISSTA'94 – International Symposium on Software Testing and Analysis, ACM Software Engineering Notes*, pp.109-124, 1994.

[6] Chow, T.S., Testing Software Design Modeled by Finite-State Machines, *IEEE Transaction on Software Engineering*, SE(4(3)), pp. 178-187, 1978.

[7] Delamaro, M.E., Proteum – A Test Environment Based on the Mutation Analysis, *Msc thesis*, ICMC/USP, São Carlos, SP, Brazil, October, 1993. (In Portuguese)

[8] Delamaro, M.E., Interface Mutation: An Inter-procedural Adequacy Criterion for Integration Testing, *PhD thesis*, IFSC/USP, São Carlos, SP, Brazil, july, 1997 (In Portuguese)

[9] Delamaro, M.E.; Maldonado, J.C.; Mathur, A.P., Interface Mutation: An Approach for Integration Testing, *IEEE Transaction on Software Engineering*, 1999 (to appear).

[10] DeMillo, R.A., Lipton, R.J., Sayward, F.G., Hints on Test Data Selection: Help for the Practicing Programmer, *Computer*, v11(4), pp.34-41, 1978.

[11] DeMillo, R.A. Mutation Analysis as a Tool for Software Quality Assurance, *in Proceedings of COMPSAC 80*, Chicago-IL, October, 1980.

[12] Fabbri, S.C.P.F., Maldonado, J.C., Delamaro, M.E., Masiero, P.C., Mutation Analysis Testing for Finite State Machine, *in proc. ISSRE'94 – Fifth International Symposium on Software Reliability Engineering*, California, pp.220-229, November, 1994.

[13] Fabbri, S.C.P.F., Maldonado, J.C., Masiero, P.C., Delamaro, M.E., Wong, E., Mutation Testing Applied to Validate Specifications Based on Petri Nets, *in proc. FORTE'95 - 8th International IFIP Conference on Formal* Description *Techniques for Distributed Systems and Communications Protocol*, Montreal, Canada, October, 1995.

[14] Fabbri, S.C.P.F., Maldonado, J.C., Sugeta, T., Masiero, P.C., Mutation Testing Applied to Validate Specifications Based on Statecharts, *in proc. ISSRE'99 – 10th International Symposium on Software Reliability Engineering*, 1999.

[15] Fecko, M.A., Uyar, M.U., Amer, P.D., Sethi, A.S., Optimum Test Sequence Generation from Estelle Specifications, *Estelle'98*, Paris, November, 1998.

[16] Henniger, O., Ulrich, A., König, H., Transformation of Estelle modules aiming at test case generation, *in proc. IWPTS'95 – IFIP International Workshop on Protocol Test Systems*, pp.45-60, Paris, 1995.

[17] ISO/TC97/SC21/WG1/DIS9074, *Estelle – A formal description* technique *based on an extended state transition model*, 1987.

[18] Lopes de Souza, W., Estelle: A Technique for the Formal Description of Services and Communication Protocols, *Revista Brasileria de Computação – SBC*, v.5(1), Rio de Janeiro, September, 1989.

[19 Myers, G., *The Art of Software Testing*, Wiley, New York, 1979.

[20] Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C., An Experimental Determination of Sufficient Mutant Operators, *ACM Transactions on Software Engineering Methodology*, v5(2), pp. 99-118, April, 1996.

[21] Probert, R.L., Guo, F., Mutation Testing of Protocols: Principles and Preliminary Experimental Results, *Protocol Test Systems, III*, Ed. by I. Davidson and D.W. LitwackNorth-Holland, pp. 57-76, 1991.

[22] Rapps, S., Weyuker, E.J., Selecting Software Test Data Using Data Flow Information, *IEEE Transaction on Software Engineering*, v11(4), pp. 367-375, April, 1985.

[23]Simão A., *Proteum-Rs/PN: A Tool to Support the Validation of Petri Net Specification Based on the Mutation Testing,, Msc thesis (in preparation)*, ICMC/USP, São Carlos, SP, Brazil, 1999. (In Portuguese)

[24] Tracz, W., Test and Analysis of Software Architectures, *in proc. ISSTA'96 – International Symposium on Software Testing and Analysis, ACM Software Engineering Notes*, pp.1-3, 1996.

[25] Ural, H., Formal Methods for test Sequence Generation, *Computer Communications*, v15(5), pp.311-325, June, 1992.

[26] Weyuker, E.; Goradia, T.; Singh, A. "Automatically Generating Test Data from a Boolean Specification", *IEEE TSE*, v. 20(5), pp.353-363, May, 1994.

[27] Wong, W.E.; Maldonado, J.C.; Mathur, A.P., Mutation versus All-uses: An Empirical Evaluation of Cost, Strength and Effectiveness, *in Proceedings of the First IFIP/SQI International Conference on Software Quality and Productivity*, Hong Kong, November, 1994.

[28] Wong, W.E., Mathur, A. P., Reducing the Cost of Mutation Testing: An Empirical Study, *J. Systems Software*, v.31, pp. 185-196, 1995.