Acta Cybernetica **15** (2002) 583–599.

Automatic Test Selection based on CEFSM Specifications*

Gábor Kovács[†], Zoltán Pap[†], and Gyula Csopaki[†]

Abstract

Mutation analysis is a fault based testing method used initially for code based software testing. In this paper, this method is applied to formal specifications and used for automatic conformance test selection. This paper defines formally a set of mutation operators for CEFSM (Communicating Extended Finite State Machine) systems to enable the automated creation of mutant specifications. Mutants of a specification are used as selection criteria to pick out adequate test cases. Two different algorithms are proposed for the generation and selection of efficient test suites. Additionally, the operators and algorithms provide the basis of an automatic tool developed at the Budapest University of Technology and Economics. We present the results of an empirical study on the well-known INRES protocol acquired using the tool.

Keywords: automatic test generation, CEFSM, mutation analysis, SDL, test selection

1 Introduction

One of the most important criteria that applies to telecommunications software is compatibility with systems from different vendors. This is usually achieved by standardization. Manufacturers of the actual products ensure compatibility by applying these specifications. At the end of the development process, the final and most important step is to test the actual products to guarantee that they work as required by the specification. This is called conformance testing, which provides the means to ensure that systems from different companies are compatible, and are able to interoperate correctly according to the standard. The test development process, however, involves significant resources: it is very time consuming and requires the

*This research is supported by ETIK. ETIK partners are: Antenna Hungaria Co., Compaq Computer Hungary Ltd., Computer and Automation Research Institute of the Hungarian Academy of Sciences, Ericsson Hungary Ltd., Hungarian Telecommunication Co., KFKI Computer Systems Co., Sun Microsystems Inc., and Westel Mobile Telecommunications Co.

[†]Department of Telecommunications and Telematics, Budapest University of Technology and Economics, Magyar tudósok körútja 2, H-1117 Budapest, HUNGARY, Phone: (36) 1-463-2225, Fax: (36) 1-463-3107, e-mail: (kovacsg,pap,csopaki)@ttt-atm.ttt.bme.hu

manual effort of many well-trained developers. Therefore, its automation is an important challenge.

Specifications may be defined either in formal or informal way. The most widely accepted technique for formal specification of telecommunications protocols is SDL (Specification and Description Language) [8]. The SDL specification of a system is an excellent starting point for both manual and automatic test case generation. It describes the behavior of the system in detail, and its graphical interface provides easy readability. Even more importantly, its formal manner makes automation possible.

Previously, there have been attempts to automatically generate test cases from SDL specifications, but all of them face the common problem of test selection. All the different approaches require a mechanism to distinguish *"good"* test cases from the unnecessary ones, possibly without any human intervention. Thus, selection criteria are needed, that can be applied automatically.

The basic idea behind the recent method is that by applying small syntactical changes, or mutations, at atomic level to the specification exactly once at a time, we intentionally produce faults [10]. The rationale is that if a test set can distinguish a specification from its slight variants, the test set is exercising the specification adequately. The erroneous specifications then can be used as selection criteria to select adequate test cases.

Mutation analysis has previously been used for code-based software verification and validation, and it has also been applied to some simple specifications [2], for example SCR (Software Cost Reduction) [7] description of software. According to the literature [14][4][16], a similar idea has been used for protocol testing, called fault-based testing. Fault-based testing requires special fault models and tries to detect faults in the implementation, with respect to the specification.

In this paper, we define methods and mutation operators especially designed to enable the automation of test selection. We do not simulate typical errors of the specification or the implementation, instead we create erroneous specifications that provide an appropriate basis for the selection of test cases.

The paper is organized as follows. In Section 2, we define the CEFSM model formally, which describes the dynamic behavior of SDL processes. Section 3 is a summary on mutation analysis. Section 4 introduces mutation operators and approaches for test selection. In Section 5,we present the outcome of the empirical analysis of the presented method using the tool developed within the confines of this research. At last, a summary is given in Section 6.

2 Extended Finite State Machines

2.1 Specification and Description Language

SDL is a formal language, widely used for specifying – especially telecommunications – systems. It has been developed by ITU-T (CCITT). One of the strengths of SDL [8] is that it is a well-accepted world standard supported by ITU-T and ISO.

Nowadays, SDL is primarily used in the telecommunications industry for the description of telecommunication protocols during the development of different hardware structures and software products, but it can be used in other fields as well. Typically complex, event-driven, real-time, and communicating systems can be effectively described in SDL.

The static structure of the system is built up hierarchically with the system object as root. The system is the formal model of an existing or planned real system. Everything not belonging to the system is called environment. Below the system level are the blocks, which can build up several levels in a tree structure. At the bottom of the hierarchy are the processes. Communication between processes is possible via signals that travel on certain predefined routes connecting processes without delay.

In our case, the most important property of SDL is that it describes the dynamic behavior of a system as a CEFSM. Processes communicating via signals specify the operation of the system. An extended finite state machine describes each process. The state machines are labeled extended, since variables and timers can also be defined. All of the processes have their own memory for storing their variables and state information, and all of them contain a FIFO buffer of infinite length that is a queue for the incoming signals.

2.2 Formal Description of Communicating Extended Automata

Formally, a CEFSM, e.g. an SDL process or system, can be described by an quintuple [12] $CEFSM = (S,\ I,\ O,\ V,\ T)$, where S, I, O and T are the finite and nonempty set of states, inputs, outputs and transitions respectively, and V is the finite set of variables.

A transition $t \in T$ is a 6-tuple: $t = (s, i, P, A, o, s')$, where $s \in S$ is the start state, $i \in I$ is an input, $P : P(V)$ is a predicate on the variables, $A : V' := A(V)$ is an action on the variables, $o \in O$ is an output and $s' \in S$ is the next state.

Initially, the configuration of the machine is represented by the initial state $s_0 \in S$ and by the initial variable values.

Inputs ($i \in I$) and outputs ($o \in O$) are communication events. They may have parameters ($I \times V$ and $I \times V$), and are realized by parameterized signals in SDL. A reception of a parameterized signal can be viewed simply as an input and a joint action assigning the new values. The length of the queue increases by one as an input arrives and the input is added to the end of the queue. The length of the queue decreases by one as an input is processed. Henceforth, $i \in I$ denotes the input to be processed, which is not necessarily the first element. The SDL specific save mechanism means that if a specific input is the firts element of the queue, then it is skipped and the next element is going to be processed.

Variables provide further details of the system's internal state. The reaction to a specified input depends on the actual value of some variables through predicates. Predicates are expressions built up from the actual subset variables at a given state. Actions represent the effect of a transition to a subset of the variables.

A FSM (Finite State Machine) is reduced if it has no inaccessible states and all states are distinguishable [1]. A state s is inaccessible if there is no input sequence that moves the machine to state s. Two states s and s' of a machine M are distinguishable if there is some input sequence Q such that executing Q from s and s' produces different output. Let us define the reduced CEFSM by extending the definition of reduced FSM in [1]:

Definition 1 (Reduced CEFSM) *Variable $v_i \in V$ is inaccessible, if $\nexists i \in I$, that after a transition $t \in T$ $v \neq A(v)$. Variable $v_i \in V$ is observable, if there are different values $x \neq y$ of the variable v_i that if $v_i = x$ then the output is o_1 after the transition $t \in T$, and if $v_i = y$ then the output is o_2 after the transition $t \in T$, and everything else is unchanged, then $o_1 \neq o_2$.*

Boolean predicate $p_i \in P$ is inaccessible if $\nexists t_1, t_2 \in T$ that in the transition t_1 $p_i(V)$ is true, and in the transition t_2 $p_i(V)$ is false. Boolean predicate $p_i \in P$ is observable, if $\exists t \in T$, that if the output after the transition t is o_1 when $p_i(V)$ is true, and the output after the transition t is o_2 when $p_i(V)$ is false, and everything else is unchanged, then $o_1 \neq o_2$.

Let us say that a CEFSM M is reduced if no states, variables and predicates of M are inaccessible, any two states of M are distinguishable, and all variables and predicates of M are observable.

3 Mutation Analysis

Mutation analysis [13] is a white-box method for developing test cases – i.e. it is based on the knowledge of the internal logic of a system. Traditional mutation analysis checks for faults in the code of programs. We, however, apply mutation analysis to specifications instead of programs, and use it as selection criteria to pick out adequate black-box test cases.

A mutation analysis system defines a set of mutation operators [11], where each operator represents a type of an atomic syntactic change. Using these operators is practical for two reasons. On the one hand, they enable the formal description of fault types. On the other hand, operators make automated mutant generation possible. By applying the operators systematically to the specification, a set of mutants can be generated.

A mutation analysis system consists of three components (Figure 1):

- Original system.
- Mutant system – it is a small syntactic variation of the original. Mutants can be created by applying mutation operators, where each operator represents a small syntactic change.
- An oracle – a person or in most cases a program to distinguish the original from the mutant by their interaction with the environment.

Traditional program-based mutation analysis assumes the competent programmer hypothesis [3]. In the current work, we assume a similar *"competent specifier"*

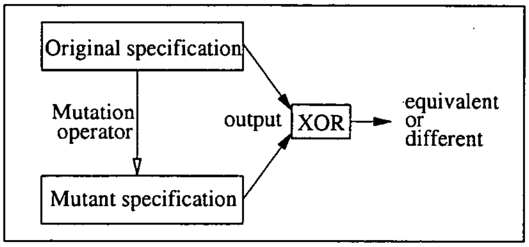


Figure 1: Components of a Mutation Analysis System

hypothesis stating that the specifier of a CEFSM system is likely to construct a specification close to the requirements, and so the test cases distinguishing syntactic variations of a specification are useful.

In the recent work, we only apply first-order faults – i.e. we apply exactly one mutation at a time –, because test sets detecting changes to the original system created by a simple error would also detect complex changes created by applying a sequence of simple mutations [13].

Test cases distinguish mutants from the original, if they produce different output. However, some of the mutants generated using the operators may be semantically equivalent to the original system. That is, a mutant and the original may compute the same function for all possible inputs. These mutants are called equivalent. All equivalents should be ignored, but all non-equivalents should be considered during test selection. Equivalence is a more complex problem in case of CEFSMs than in case of FSMs.

Definition 2 (Equivalence) *Let M_1 and M_2 be two CEFSMs with known structure, and identical input and output alphabet. Homeomorphism from M_1 to M_2 is the mapping Φ from S_1 to S_2 and the mappings Ψ_V, Ψ_P and Ψ_A from V_1 to V_2, P_1 to P_2 and A_1 to A_2 respectively, such that $\forall s_1 \in S_1$ and $\forall i \in I$ it is true for transition $t_2 = (\Phi(s_1), i, \Psi_P(P_1), \Psi_A(A_1), o_2, s_2'), t_2 \in T_2$ and transition $t_1 = (s_1, i, P_1, A_1, o_1, \Phi(s_1')), t_1 \in T_1$ that $s_2' = \Phi(s_1')$, $V_2' = \Psi_V(V_1')$ and $o_2 = o_1$.*

If Φ and Ψ are bijections, then it is an isomorphism. In this case, M_1 and M_1 are equivalent machines.

Definition 3 (Pseudo-equivalence) *Let M_1 and M_2 be two CEFSMs with known structure, and identical input and output alphabet. M_1 and M_1 are pseudo equivalent machines, if there exists a bijection Φ between S_1 S_2 such that $\forall s_1 \in S_1$ and $\forall i \in I$ it is true for transitions $t_2 \in T_2$ and $t_1 \in T_1$ that $o_2 = o_1$ if $s_2 = \Phi(s_1)$.*

Figure 2 shows the relationship among equivalents, pseudo-equivalents and mutants. We can make the following statements:

- The specification we investigate is an element of the set of equivalents *Spec* $\in$ *EQ*.
- The set of equivalents is a subset of pseudo-equivalents $EQ \subseteq PE$.

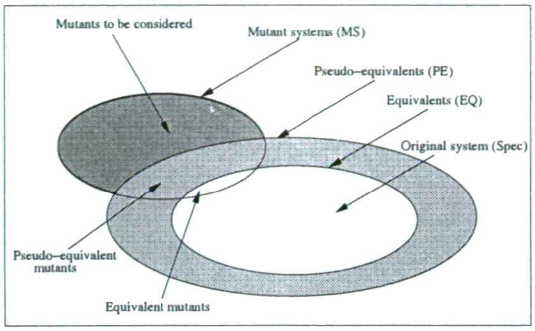


Figure 2: Equivalence Relationships (based on [16])

- A pseudo-equivalent mutant is an element of the $MS \cap PE$ set.
- An equivalent mutant is an element of the $MS \cap EQ$ set.

When applying the recent method, only mutant in the set $MS \setminus (MS \cap PE)$ should be considered.

4 Test Selection Method

4.1 Mutation Operators Proposed for CEFSM Specifications

In the recent paper, a very important consideration for the definition of operators is, that they should be simple enough to enable the automation of the mutation testing process, in order to design a tool. And additionally, the properties of SDL should also be taken into account. Operators should create finite – and as small as possible – number of mutant specifications. If possible, operators should not create any pseudo-equivalents and – of course – minimize the number of equivalents, because pseudo-equivalent systems unnecessarily increase the time required for the test case selection.

According to these considerations, mutation operators we use model atomic faults, and only one fault at a time. That is, we modify exactly one element in a transition $t \in T$. To select conformance test cases, it is essential to generate syntactically correct mutants. Syntactic correctness is necessary to be able to execute the mutant system. Some types of semantic errors should be detected, according to ([12]), since for example after applying an operator some states may become unreachable, preventing conformance testing. Such mutant systems should be detected and dropped during a semantic analysis phase.

We defined five classes of mutation operators for CEFSM descriptions according to which part of the automaton they are applied to:

- state modification operators,
- input modification operators,

- output modification operators,
- action modification operators,
- predicate modification operators.

Additionally, there is a specific operator for the mutation of save statements in SDL.

It is important to note, that we replace non-boolean predicates with a sequence of boolean predicates, and apply the operators on them. For the types of mutations when a component of the machine is replaced by an other component, instead of creating all possible combinations, it is sufficient to only produce one.

Henceforth, let the $\Omega()$ function represent the syntactical change applied.

Operator 1 (State) Modifying states. Here only the exchange of inputs should be considered. The mutation operator is applied either to the actual or to the next state:

1. Mutating the next state in transition $t \in T$: $t = (s, i, P(V), A(V), o, \Omega(s'))$. In this case we replace the next state, that is we lead the system to a wrong state and induce incorrect operation.
2. Mutating the actual state in transition $t \in T$: $t = (\Omega(s), i, P(V), A(V), o, s')$. This operator replaces the transition function of two states.

The mutation of the stating state is a special case of state mutation, where s_0 is modified: $\Omega(s_0)$.

Operator 2 (Input) The input mutation in the transition $t \in T$: $t = (s, \Omega(i), P(V), A(V), o, s')$.

1. Using $\Omega(i) := null$. This mutation is equivalent with the removal of a transition branch for an input at a given state.
2. Using the transition of input $i_x \in I'$, where $I' \subseteq I$ is the set of inputs, that have explicit transitions defined at the given s state. This means that we exchange the transition of two inputs.
3. Assigning the transition of input i_{inopp} ($i_{inopp} \in I$, but $i_{inopp} \notin I'$, where $I' \subseteq I$ is the set of inputs, that have explicit transitions defined at the given s) to the existing transition branch of input $i \in I$. This mutation means that we add a transition branch for the input i_{inopp}, that was implicitly consumed previously. Using this mutation, also the processing of inopportune (valid input arriving at wrong time) inputs can be inspected.
4. As mentioned previously, inputs and outputs may have parameters. If $i \in I \times V$, then we can mutate not only the input symbol, but the input parameter leaving the input symbol unchanged. This type of mutation is practically an action mutation, and can be viewed as the mutation of the implicit action assigning the new values. In this case, $\Omega(i) = i(\Omega(v))$, where $\Omega(v) := null$ – according to the action mutation (see below).

Operator 3 (Output) The mutation of an output event in the transition $t \in T$ is: $t = (s, i, P(V), A(V), \Omega(o), s')$. If the output symbol has parameters ($o \in O \times V$), then we have the possibility to modify the parameter: $\Omega(o) = o(\Omega(v))$, where $\Omega(v) := null$.

Operator 4 (Action) It is difficult to define a general mutation operator for actions, because of the presence of abstract data types. Only the $\Omega(A(V)) := null$ operator, that is the deletion of an action is suitable for this case. Missing action operator: $t = (s, i, P(V), \Omega(A(V)), o, s')$.

Operator 5 (Predicate) The mutation of boolean predicates has similar effects as the mutation of inputs.

1. Exchanging two branches of a decision in the transition $t \in T$ can be done simply negating the whole expression $t = (s, i, \Omega(P(V)), A(V), o, s'$, where $\Omega(P(V) := not(P(V))$.
2. Setting the predicate to be stuck-at-true ($\Omega(P(V)) := true$) or stuck-at-false ($\Omega(P(V)) := false$) brings on the removal of the other branch.

Operator 6 (Save) Since this paper consideres SDL to be the specification language used, we have to take its specialities, like the save mechanism (see Section 2.2), into account. Note that save is not part of the CEFSM model. This operator removes the save statement.

In Table 1, examples show some of the mutation operators and their realization in the context of SDL.

Operators	Original	Mutant
State	`NEXTSTATE wait;`	`NEXTSTATE connected;`
Input	`INPUT ICONresp;`	`INPUT IDISreq;`
Output	`OUTPUT CC;`	`OUTPUT DT (number, d);`
Action	`counter := 1;`	`/* Missing */`
Predicate	`DECISION sdu!id = CC;`	`DECISION NOT(sdu!id = CC);`
Save	`SAVE IDATreq (d);`	`/* Missing */`

Table 1: Mutation Operators for SDL

4.2 Theoretical Analysis of the Operators

Since the operators are defined in a formal manner, the analysis of their relationship is possible. We can show that there are correlations among the operators.

Theorem 1 (Relation between the brach exchange and the stuck at operator) *Applying the stuck-at (Ω_{SA}) and the branch exchange (Ω_{BX}) operators to the same predicate, then for the detection $\Omega_{SA} \rightarrow \Omega_{BX}$.*

Proof (based on [11]).
Let $p \in \mathcal{P}$ be the predicate to be mutated. The theorem follows: $\Omega_{SAfalse}(p) \rightarrow \Omega_{BX}(p)$ and $\Omega_{SAtrue}(p) \rightarrow \Omega_{BX}(p)$.
In case of deleting the true branch the detection condition is $p \oplus \Omega_{SAfalse}(p) = p \oplus false = p$.
In case of exchanging the branches the detection condition is $p \oplus \Omega_{BX}(p) = p \oplus not(p) = true$. That is, we always detect the branch exchange mutation. Since $p \rightarrow true$, then $\Omega_{SAfalse} \rightarrow \Omega_{BX}$.
In case of deleting the false branch the detection condition is $p \oplus \Omega_{SAtrue}(p) = p \oplus true = not(p)$. Since $not(p) \rightarrow true$, then $\Omega_{SAtrue} \rightarrow \Omega_{BX}$.
Therefore $\Omega_{SA} \rightarrow \Omega_{BX}$.

We made the following findings and statements concerning the operators:

- The mutation operator for states and next states are equivalent, since the tail state of a transition is the initial state of the next: $t_1 = (s, i, p(V), a(V), o, \Omega(s')) \Leftrightarrow t_2 = (\Omega(s'), i', p(V'), a(V'), o', s'')$.
- Timeouts in the CEFSM systems (e.g. SDL) can be considered simple inputs, and accordingly, the input mutation operator mutates them. To be able to test timer transitions, timeout events are made controllable from the environment. That is, whenever a test case reaches a timeout (for example `timeout T3;` in an MSC (Message Sequence Chart) test case), a corresponding input is sent directly to the owner machine of the timer from the environment, and after its consumption the corresponding timer transition is executed. During the test execution, in the test case a timeout explicitly indicates that the tester has to wait for the duration of the timer (e.g. "`Timeout T3`" and "Timer T3 shall be in the range 1 sec to 1.5 secs."). This way, methods for test case selection in the next section (4.3) become time independent.
- Definition 1 implies, that if using a reduced CEFSM, state exchange mutation and branch exchange for the predicates do not create equivalents, therefore, it is advised to use reduced specifications for mutant generation.

4.3 Algorithms for Test Generation

We defined two approaches for test generation. Although they are similar, being built on the same concept, they produce different resulting sets. Both approaches require the CEFSM specification of a system. There are practical and widely used tools, to assist the specification process like Telelogic Tau [15]. After the specification is completed, processing in both cases can be all automated.

We represent test cases in MSC (Message Sequence Chart) [9] [6]. The MSC test sets can later easily be transformed to different test description languages for example TTCN (Tree and Tabular Combined Notation). Tools support the semi-automatic TTCN table generation from MSC test case specifications.

Algorithm 1 (Deriving test cases from a specification) Figure 3 shows our first approach consisting of the following steps:

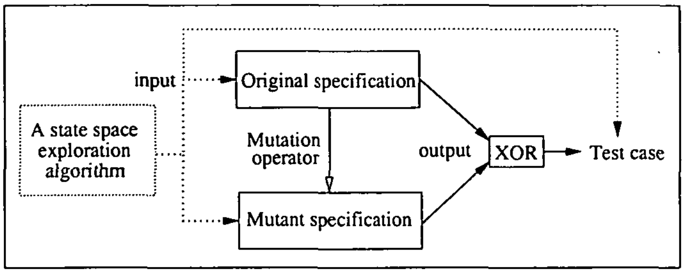


Figure 3: Mutation Analysis of CEFSMs Using a State Exploration Algorithm

1. Apply a mutation operator to the machine, that is, create a mutant specification.
2. Use a state space exploration algorithm to compare the mutant with the original system. Of course, we only stimulate the system using inputs from the environment, and only check the outputs to the environment for inconsistency. This is because our intent is to create test suites for black-box testing. In most cases, we should explore the state space of the original system, but in case of certain operators the desired result can be achieved by exploring the mutant state space. The algorithm must have break conditions, e.g. reaching a certain depth, exceeding a time limit etc.
3. When the state space exploration algorithm finds an inconsistency, it generates a test case based on the set of stimuli sent from the environment and the outputs received until the inconsistency was discovered.
4. Repeat steps 1-3 for all possible mutants until corresponding test cases are generated.
5. As a result we get a set of test cases.

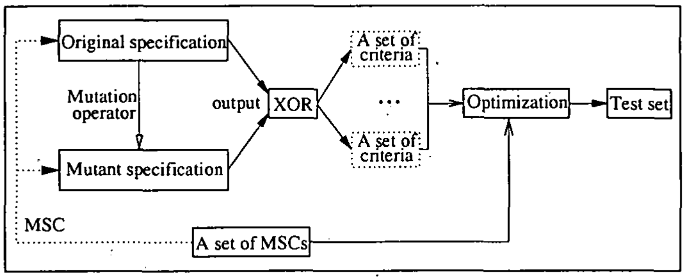


Figure 4: Mutation Analysis of CEFSMs Based on an Existing Test Set

Figure 4 shows our second scenario. Here we assume that a finite size, unstructured and highly redundant test set exists, for example in a form of a set of MSC test cases. These test sets can be created by the means of state space exploration algorithms exploring the specification of the system. If we apply mutation operators checking inopportune inputs, this initial set of test cases also has to include inopportune test cases.

Let the matrix of criteria $\mathbf{C}$ be a two-dimensional matrix with boolean values.

Algorithm 2 (Selecting test cases from an existing set) Our second approach consist of the following steps:

0. Create a set of test cases.
1. Apply a mutation operator to the system, that is, create a mutant specification (the i^{th} mutant).
2. Run the test set on the mutant specification and check for inconsistency.
3. Create a row vector $\mathbf{C}_i$ (the i^{th} row of the C matrix).
   - Let $\mathbf{C}_{ij}$ be 0, if the j^{th} test case is not able to detect the i^{th} mutant.
   - Let $\mathbf{C}_{ij}$ be 1, if the j^{th} test case is able to detect the i^{th} mutant.

   In other words, select the test cases from the original set, which kill the given mutant.
4. Repeat steps 1-3 for i=1 to N, where N equals the number of all possible mutants that can be created from the given system using the mutation operators defined in Section 4.1.
5. Acquire the matrix of criteria, where rows represent the mutants and columns represent the test cases in the original set. This matrix describes for each test case the mutations the given test case is able to detect.
6. Apply some simplifications to the matrix of criteria (C):
   - If there is a column $\mathbf{C}_j$ in C, where $\forall i : \mathbf{C}_j[i] = 0$, it represents that the j^{th} test case did not find any of the mutants. Therefore, the j^{th} column can be omitted.
   - If there is a row C_i in C, where $\forall j : \mathbf{C}_i[j] = 0$, it represents either that the i^{th} mutant is an equivalent, or that there was no test case in the original set that could find the difference (kill the mutant).
   - If there are rows $\mathbf{C}_m$ and $\mathbf{C}_n$ in $\mathbf{C}$, where $\forall j : \mathbf{C}_m[j] \leq \mathbf{C}_n[j]$, then the row $\mathbf{C}_m[j]$ is unnecessary.
7. Select an optimal test suite from the original set, using an integer programming method.

We can also automatically assign weights to the test cases, implicating how time-consuming they are. Thus, we can further improve the efficiency of the selection.

Both algorithms have their advantages and drawbacks. The first algorithm requires less computation and time, but is it does not provide enough data for an optimization algorithm. An important benefit of the second – and more complex – method is that the optimization process ensures that only adequate test cases will be selected, and thus, a more effective test set is created. A major drawback of the second method is that it is quite computation-intensive. If the two algorithms are executed consecutively, the first algorithm provides a rough set of – MSC – test cases, and then the second algorithm further improves the efficiency of the test set.

5 An experiment on the system INRES

5.1 The INRES system

We used the well-known sample telecommunications system INRES ([5]) to investigate the method presented (see Figure 5). We chose a sample protocol that includes all the typical properties of real life protocols. It is built on the OSI concept, and it contains some basic OSI elements. INRES is a connection-oriented protocol that operates between two protocol entities Initiator and Responder. These protocol entities communicate over a Medium service. Although it is not a real protocol, its specification contains most of the syntactic elements of SDL. The SDL specification of the system was created using the Telelogic Tau ([15]). The structure of the system and the states of the processes are shown in Figure 5. (The transitions of the processes are shown schematically, dots represent decisions. Inputs, outputs and actions are not represented. For more details see [5].)

Table 2 summarizes the syntactic properties of the six processes of the INRES system – described in SDL. The columns represent the following:

- *States* – number of states in the given process.
- *Inputs* – the sum of processed inputs in each state of the given process.
- *Outputs* – the sum of generated outputs in each transition of the given process.
- *Decisions* – the number of boolean decisions (i.e. the answer is either true or false) in all transitions of the given process.
- *Saves* – the number of save statements in the given process.
- *Transitions* – the number of different transitions in the given process.

Processes	States	Inputs	Outputs	Tasks	Decisions	Saves	Transitions
Initiator	4	10	9	8	4	1	14
CoderIni	1	3	4	5	2	0	5
Responder	3	5	7	2	1	0	9
CoderRes	1	4	3	6	1	0	5
MSAPMan1	1	2	2	0	0	0	2
MSAPMan2	1	2	2	0	0	0	2

Table 2: Syntactic Properties of the Processes in INRES

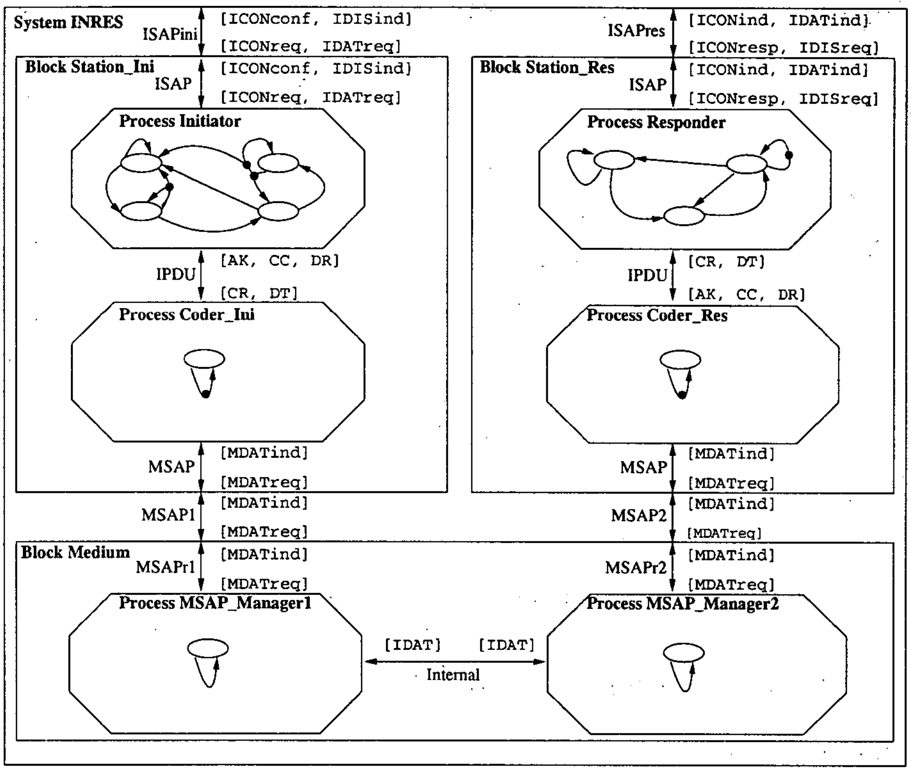
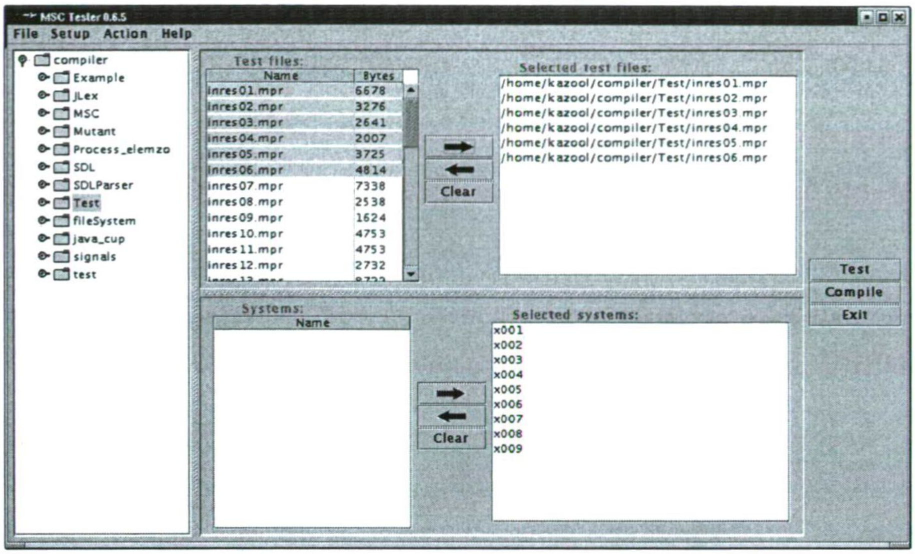


Figure 5: The INRES system in SDL

5.2 The tool

The tool consists of several components. One of the inputs of the tool is a textual SDL specification. First, this specification is modified, so that timers are made controllable, and decisions with multiple branches are transformed to series of boolean decisions. These transformations do not affect the behavior of the system. An other component of the tool applies mutation operators to this specification, and generates a set of mutant specifications. A semantic check is applied to this set to find critical semantic errors. The next component generates program code from both the original and the mutant specifications. The test execution component is the core of the tool. This implements the second algorithm. The inputs of this component are the program codes generated before and MSC test cases given in textual form. This component outputs a boolean matrix, which is the input of the last component, the optimizer. The final output of the tool the names of the MSC test cases found adequate.



Name	Bytes
inres01.mpr	6678
inres02.mpr	3276
inres03.mpr	2641
inres04.mpr	2007
inres05.mpr	3725
inres06.mpr	4814
inres07.mpr	7338
inres08.mpr	2538
inres09.mpr	1624
inres10.mpr	4753
inres11.mpr	4753
inres12.mpr	2732

```
/home/kazool/compiler/Test/inres01.mpr
/home/kazool/compiler/Test/inres02.mpr
/home/kazool/compiler/Test/inres03.mpr
/home/kazool/compiler/Test/inres04.mpr
/home/kazool/compiler/Test/inres05.mpr
/home/kazool/compiler/Test/inres06.mpr
```

Name

```
x001
x002
x003
x004
x005
x006
x007
x008
x009
```

Figure 6: Mutation tester for SDL and MSC

5.3 The experiment and empirical analysis

We generated 87 MSC test cases using Telelogic Tau for the INRES protocol. This set intentionally included some appropriate and some useless test cases, and most of them were generated randomly. By applying the operators to the specification, 118 mutants were generated automatically. Finally, nine test cases of the original set were selected. The selection included some of the test cases we initially considered appropriate based on our knowledge of the system.

The resulting test set depends on the number and the quality of test cases in the original set. Therefore, if test cases in the initial set are generated automatically, then the resulting set is only influenced by their number. The selected test cases achieve 100% symbol coverage using the Telelogic Tau, which also indicates that they are, in fact, adequate.

This whole limited experiment took using the second algorithm on a computer with a PIII/500MHz processor and 256 Mbytes of RAM 103 minutes. The execution time – of both algorithms in Section 4.3 – can be decreased by dividing the test set and executing the groups in parallel. The time required to optimize an existing set – using the second algorithm – is proportional to the number and to the length of test cases and to the size of the specification.

Table 3 shows the data acquired during the experiment.

As the data show, twenty mutants have not been discovered by any of the MSC test cases. The worst mutation uncover ratio is, in the case of state mutation operators, and the best is in the case of input exchange. More than half (12) of the cases, where no difference has been found is in the Initiator process that is the

Mutation operators	Mutants generated	Detected	Detection ratio [%]
State	24	14	58
Input	28	28	100
Output	36	33	92
Action	21	15	71
Predicate	8	7	87
Save	1	1	100

Table 3: Mutation Analysis Applied to the SDL specification of INRES

largest in the system. Six of the mutants of the Responder process have not been killed. Both in case of processes CoderIni and CoderRes, one mutation has not been revealed.

There have been some mutations discovered by only one test case, we call them critical mutants. The test cases detecting these critical mutants have to be included in the resulting set. Interestingly, these test cases give eight of the nine selected. Out of the eight critical mutants, two have been generated using predicate, two using output, three using state and one using action mutation operator. Applying input mutation generated no critical mutants.

Both this, and the mutation uncover ratio indicate that input mutation (and input like mutations, e.g. missing transition) operators result in very rough mutant systems. That is, they produce radical changes in the behavior of the – INRES – system that can be detected by most of the test cases. State mutation, on the other hand, produces errors that can be discovered by only a small percentage of the test cases, but it has provided three critical mutants, more than any other operator.

6 Conclusions

Conformance testing is a vital part of the standard based telecommunications protocol development process. In the practice, the creation of test suites is usually a very time consuming manual process, even though several computer aided test generation methods have already been developed. By means of the method proposed, in this paper, it is possible to automate all steps of the test selection process, and to create effective test suites.

In this paper, we described how to apply mutation analysis, a white box method, to formal SDL specifications, and use mutant systems to automatically select adequate test cases for black box testing. For this purpose, we created and formally specified a set of mutation operators for CEFSM and SDL specifications. We also presented two slightly different algorithms for automatic test selection using the operators. We investigated empirically the mutation operators used. The recent method and the tool is useful not only for simple protocols, but also for real, complex telecommunications systems described in SDL.

In the future, we plan to make more experiments to reveal the effect of using different initial test and operator sets.

Acknowledgements

We would like to extend our gratitude to Zoltán Andriska, Gábor Bátori, Dung Le Viet and Antal Wu-Hen-Chang for their contribution in the tool development.

References

[1] A. V. Aho and J. D. Ullman. The Theory of Parsing, Translation, and Compiling, volume 1 of *Automatic Computation*. Prentice-Hall, 1972.

[2] P. E. Ammann and P. E. Black. *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, 1999.

[3] P. E. Ammann, P. E. Black, and W. Majurski. *Using Model Checking to Generate Tests from Specifications*. In *Second IEEE International Conference on Formal Engineering Methods*, pages 46–54, 1998.

[4] C. Bourhfir, R. Dssouli, and E. M. Aboulhamid. *Automatic Test Generation for EFSM-based Systems*. http://citeseer.nj.nec.com/114451.html.

[5] J. Ellsberger, D. Hogrefe, and A. Sarma. SDL Formal Object-oriented Language for Communicating Systems. Prentice Hall, 1997.

[6] J. Grabowski, D. Hogrefe, and R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. North Holland, 1993.

[7] K. L. Heninger. *Specifying Software Requirements for Complex Systems. IEEE Transactions on Software Engineering*, 6(1):2–13, January 1980.

[8] ITU-T. *Recommandation Z.100: Specification and Description Language*, 1992.

[9] ITU-T. *Recommandation Z.120: Message Sequence Chart*, 1996.

[10] D. R. Kuhn. *A Technique for Analyzing the Effects of Changes in Formal Specifications. The Computer Journal*, 35(6):574–578, 1992.

[11] D. R. Kuhn. *Fault Classes and Error Detection in Specification Based Testing. ACM Transactions on Software Engineering Methodology*, 8(4), October 1999.

[12] D. Lee and M. Yiannakakis. *Principles and Methods of Testing Finite State Machines – A Survey. Proc. of the IEEE*, 43(3):1090–1123, 1996.

[13] R. A. De Millo, R. J. Lipton, and F. G. Sayward. *Hints on Test Data Selection: Help for the Practicing Programmer. IEEE Computer*, 11(4):34–41, April 1978.

[14] Alexandre Petrenko, Gregor von Bochmann, and Ming Yu Yao. *On Fault Coverage of Tests for Finite State Specifications*. In *Computer Networks and ISDN Systems*, volume 29, pages 81–106, 1996.

[15] Telelogic Tau. http://www.telelogic.com.

[16] C.-J. Wang and M. T. Liu. *Generating Test Cases for EFSM with Given Fault Model.* In *INFOCOM 93*, volume 2, pages 774–781, 1993.