The SESAME Experience: from Assembly Languages to Declarative Models

Yves Crouzet, Hélène Waeselynck, Benjamin Lussier, David Powell
first-name.last-name@laas.fr
LAAS-CNRS, University of Toulouse
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4 – France

Abstract

SESAME (Software Environment for Software Analysis by Mutation Effects) is a fault injection tool using mutation as the target fault model. It has been used for 15 years to support dependability research at LAAS-CNRS. A salient feature of SESAME is that it is multi-language. This made it possible to inject faults into software written in assembly languages, procedural languages (Pascal, C), a data-flow language (LUSTRE), as well as in a declarative language for temporal planning in robotics. This paper provides an overview of the tool, and reports on its use in experimental research addressing either fault removal or fault tolerance topics.

1. Introduction

SESAME (Software Environment for Software Analysis by Mutation Effects) is a mutation environment developed by the Dependable Computing and Fault Tolerance Group at LAAS-CNRS.

Central to the research of the group is the notion of *fault*, yielding a structuring of our activities according to the four means to attain dependability [4]:

- *Fault prevention* to prevent the occurrence or introduction of faults.
- *Fault tolerance* to avoid service failures in the presence of faults.
- *Fault removal* to reduce the number and severity of faults.
- *Fault forecasting* to estimate the present number, the future incidence, and the likely consequences of faults.

In this context, fault injection may be used as an experimental technique to assess the effectiveness of fault removal methods, to estimate the consequences of faults on a target system, or as a testing method for removing development faults from fault-tolerance mechanisms. Indeed, the group has a long tradition of developing fault injection tools. The target faults span from hardware faults injected at the pin level [1], into VHDL models [2] or in memory [3], to software faults injected into the source code [5, 6] or at the API of off-the-shelf components [14]. SESAME falls into the category of software fault injection tools, using source code mutation as the target fault model. It was originally developed to support research on software testing, which corresponds to the classical usage of mutation environments [9, 22]. Currently, it is being used in research on fault tolerance mechanisms for autonomous robot systems.

A salient feature of SESAME is that it is multi-language. This made it possible to inject faults into software written in assembly languages, procedural languages (Pascal, C), a data-flow language (LUSTRE), as well as in a declarative language for temporal planning in robotics. The counterpart of the multi-language facility is the comparative simplicity of the supported mutations: SESAME can only produce mutants that are based on search/replace patterns. When more sophisticated mutation operators are needed, other tools must be used[1]. Despite this limitation, SESAME has proven a useful experimental support. Since its first version fifteen years ago, it has been used in the framework of several research activities. Today, it is still part of our fault injection toolbox.

This paper provides an overview of the SESAME experience. Section 2 describes the tool architecture. It is explained how the kernel is kept completely independent from the target language. Also, SESAME can be easily parameterized to use external facilities like commercial test tools, or user-supplied programs (to address specific experimental needs). Section 3 reports on the use of SESAME in dependability research, including both fault removal and fault tolerance topics. Concluding remarks are provided in Section 4.

[1] For example, the JavaMut tool (also developed in our group) implements mutation operators based on syntactic analysis and reflection [5].

2. The SESAME environment

Figure 1 provides an overview of the architecture of SESAME. It is composed of two main modules described in Sections 2.1 and 2.2:

- The *Mutant Generator*, which produces a database of mutants for the target program(s).
- The *Mutant Runner*, which executes the mutants with given test sets.

These generic modules can be tuned to specific experimental settings by associating them with external utilities like language compilers, post-processing tools for automating the analysis of raw results, or commercial test tools. This is illustrated in Section 2.3 by showing the coupling with the ATTOL test tool (now IBM Rational Test RealTime).

SESAME runs on standard Unix workstations. The modules are called via simple Unix commands. They are parameterized by supplying a configuration file.

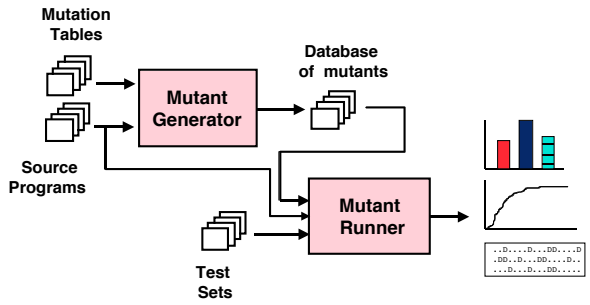


Figure 1. Overview of SESAME

2.1 The Mutant Generator

In contrast to other environments (e.g., Mothra [15] or Proteum [8]), SESAME does not offer a *fixed* set of mutation operators defined at the *syntax level*. The user may *edit* tables to enter the description of *string-level* operators. When these operators are applied to a target source program, a database of mutants is created. It is stored as a single file containing the compact representation of all mutants, so as to minimize disk space. Technically, the compact representation is the output result of a `diff` command. It allows the mutant to be re-generated from the original source program. Note that the `diff` output formats are rich enough to efficiently accommodate changes affecting several lines of code. Hence, it is possible to supplement the database of mutants by additional files describing more complex faults. We used this possibility to perform experiments involving both SESAME-generated mutations and real faults.

Typically, the generator uses three different tables, each grouping operators from a given category:

- Numerical value replacement.
- Operator replacement.
- Symbol replacement (constants, variables, or functions).

In each case, the operators are applied using simple search/replace procedures. Since there is no guarantee that a produced mutant is syntactically correct (which would involve a more sophisticated implementation with preliminary parsing), it has to be validated before being stored in the database. The usual validation is to try to compile the mutant using an external compiler. If the mutant successfully compiles, and the resulting object code is different from the one of the original program, then the mutant is stored.

The main advantage of this crude procedure is that the generation module can be used whatever the source language. Tackling a new target only requires the mutation tables to be adapted. Generally speaking, tables working on numerical values or operators are language specific, while tables working on symbols are more specific to a target program.

Table 1 shows examples of mutation operators on integer values (here, their definition is adapted to the C language). Applying these operators consists of finding any constant value, and then of appending the corresponding string to it (e.g., `10` is replaced by `10<<1`).

Operator	Comments
`<<1`	`Multiplication by 2 (left shift)`
`>>1`	`Division by 2 (right shift)`
`+1`	`Adds 1`
`-1`	`Subtracts 1`
`^1`	`Flips the bit having position 0`
`^2`	`Flips the bit having position 1`
`~`	`One's complement`

Table 1. Mutations on integer values

Table 2 exemplifies mutation operators working on language operators, or on symbols. They are implemented by search/replace procedures on strings. Whenever the target string (on the lefthand side of a unidirectional arrow) is found in the source code, it is successively replaced by all alternative strings (on the righthand side of the arrow). The bidirectional arrow is used as shorthand for the list of unidirectional replacements obtained by permuting the lefthand string with every righthand string. For example,

```
str1 <---> str2, str3
```

is shorthand for:

```
str1 ---> str2, str3
str2 ---> str1, str3
str3 ---> str2, str1
```

In this way, each element of the set {`str1`, `str2`, `str3`} is substituted for every other element.

Note that, for permissive languages like C, it may be desirable to replace an element by another element having a different type. This is exemplified by the third definition in Table 2, mixing comparison and assignment operators (indeed, confusing '==' and '=' is a typical bug for C programmers).

Operator	Comments
* ---> /, +, -	Substitution of arithmetic operators
> <---> <, <=, >=, ==	Substitution of comparison operators
== <---> =	Substitution of comparison and assignment
var1 <---> var2	Substitution of variable names
true <---> false	Substitution of constant names

Table 2. Mutations on operators and symbols

The pseudo-algorithm in Figure 2 gives a (much) simplified view of the generation of mutants. The real generation module involves 900 lines of Pascal code. The pseudo-algorithm shows how the operations performed by the module are combined with operations performed by external utilities (indicated in italic characters). Two external utilities are called:

- A *compiler for the target language*, which allows the object code of the original program to be produced.
- A *mutant validation program*, which decides whether the current mutant should be stored in the database.

A minimal mutant validation program is supplied by SESAME. As already explained, it tries to compile the mutant, and compares its object code to that of the original program. The latter operation aims to reject trivially equivalent mutants. Obviously, the crude comparison of the object codes is by no way sufficient for the detection of equivalent mutants. This is why mutant validation is provided as an external facility: this leaves open the possibility of using advanced tools for the detection of equivalence, as far as such tools are available for the target language. We will come back to this problem in the conclusion of the paper. In practice, manual detection was used in the framework of all our experiments.

```
read configuration file;
call compilation program with original source;

for each line in a mutation table
do
  for each target string occurrence (or each
  integer value) found in original source
  do
    for each replacement string
    do
      introduce mutation in the source;
      call mutant validation program;
    done ;
  done;
done ;
```

Figure 2. Simplified view of the mutant generator

2.2 The Mutant Runner

The mutant runner allows mutation analysis to be conducted on one or several test sets. Each mutant stored in the database is recreated, using the original program source and the compact mutation description. After compilation and linking, the mutant is executed with the test set(s). The outputs are compared to reference outputs for that test set, in order to determine whether the mutant was killed. Some post-processing is also performed according to the desired assessment measures.

Figure 3 gives a simplified view of the mutant runner. As in the case of the mutant generator, the design was driven by the need to easily adapt to various experimental settings. Flexibility is provided by making extensive use of external programs (indicated in italic characters). The programs are typically simple shell scripts (about 10 lines), which allows interfacing with more complex utilities, depending on the experimental needs. Figure 3 shows six categories of programs:

- The *opening session program* may, e.g., create directories for temporary files, import a local copy of the original source program, etc.
- The *preprocessing of a test set* typically consists in preparing the test environment for that set (drivers, reference outputs, etc.).
- The *compilation/link program* makes the executable of the mutant.
- The *mutant execution program* exercises the mutant with a target test set and records the outputs.
- The *postprocessing of a test set* determines whether the mutant was killed. More detailed information (e.g., the number of revealing inputs in the set) may also be produced.

```
read configuration file;
call opening session program;

for each test set
  call preprocessing test set program;

for each mutant
do
  read reduced description in database;
  generate complete source of mutant;
  call compilation/link program;
  for each test set
  do
    call mutant execution program;
    call postprocessing test set program;
   done ;
done ;

call closing session program;
```

Figure 3. Simplified view of the mutant runner

- The *closing session program* performs some final postprocessing, e.g., to produce statistics on the whole set of experiments. It also cleans up the working environment by removing the temporary files.

The external programs may be used to interface SESAME with commercial test tools. This will be exemplified in the next section by the ATTOL tool. In the remainder of this section, we provide examples of postprocessing utilities that were developed to support our experimental needs.

Figure 4 shows the postprocessing of a test set. The first line of the generated report provides statistics on the effectiveness of the set with respect to a target mutant. The mutant was killed by 8 inputs, representing 18.6% of the set. The first revealing input was the 12th in the set (indicated by the number in brackets). Then, a detailed trace is provided: '`D`' identifies revealing inputs, while '`.`' denotes non-revealing ones. Such reports proved quite useful for analyzing the revealing power of test sets with respect to specific faults. Also, the statistics were an indicator of the quality of the database of mutants. It could be checked whether a significant number of mutants were killed by few inputs, even in large test sets. The detailed trace was useful for identifying error bursts due to memory effect.

Figures 5 and 6 provide two examples of postprocessing launched by the closing session program.

Figure 5 plots the evolution of the mutation scores versus the number of executed inputs. The observed evolution, a rapid growth followed by a sharp slowdown, is typical. The test data rapidly uncover the "easy" mutants during a first phase. Then, the incremental gain becomes almost nil because the remaining mutants are those that are difficult to kill by the adopted test strategy. A change in the strategy can be identified by stages in the plot (see, e.g., the evolution observed at N=85 for test sets 2 and 3). Note that if the score has not yet stabilized at the end of the test set, or before a change in the strategy, this probably means that the test size is insufficient. Hence, the analysis of the plot may be interesting to determine stopping criteria. The evolution of the mutation scores also gives information on the comparative efficiency of test sets (e.g., although the final scores are similar, test set 1 turns out to be more efficient than the others).

Figure 6 shows information produced by the SESAME `mcov` utility (named after `tcov`, a Unix utility for analyzing structural coverage). Like `tcov`, `mcov` annotates the source code with coverage information. Figure 6 reports from experiments involving three test sets. Each block of instructions in the original source program is annotated by:

- The number of times the block was executed by each test set. For example, Block 2 was executed respectively 12, 10 and 6 times by test sets 1, 2 and 3.
- The number of mutants generated for each line of the block. For example, 12 mutants were generated for the line "`SUM = VAL1 + VAL2;`".
- Information on the mutants not killed by at least one test set (report starting by '`*`'). For each live mutant, the report consists of (i) the mutant id and the list of non revealing sets, (ii) the description of the mutation. For example, in Block 3, the mutant with id 74 was not killed by any of the test sets. The mutation replaced the `MAX` symbol by `65536`.

The `mcov` utility may serve two purposes. First, it may be used to check whether the mutants are homogeneously spread over the source code. Second, it facilitates the analysis of the live mutants. If the test sets did not kill a number of mutants affecting the same location, then there might be some common causes.

```
Mutant killed by 8 inputs (12) 18.6%

..........D......D.D......DD......DD....D.
```

Figure 4. Example of test set postprocessing

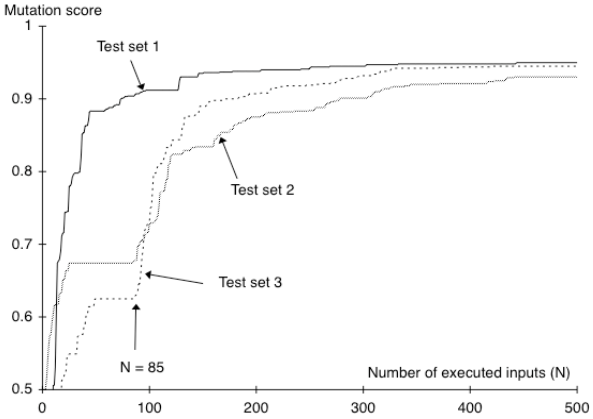


Figure 5. Evolution of the mutation scores

```
....
#Block 2 : struct. cov. sets = 12 10 6
    12    SUM = VAL1 + VAL2;
     8    if SUM > MAX
#Block 3 : struct. cov. sets = 4 5 2
     8    SUM = MAX;
* ==> (74)  1 2 3
*         SUM = 65536;
* ==> (76) 3
*         SUM == MAX ;
....
```

Figure 6. Coverage analysis by `mcov`

2.3 Coupling SESAME with ATTOL

A transfer of SESAME to industrial partners motivated the coupling with ATTOL. In an industrial context, SESAME was intended to be used to assess the quality of test sets, in addition to the usual structural coverage assessment supplied by ATTOL.

ATTOL (now IBM Rational Test RealTime) is a commercial tool for unit and integration testing. It is based on a language (the *ATTOL language*) allowing the specification of *test plans* (involving several test scenarios). The ATTOL preprocessor generates a test program from the plan. The test program is then compiled and linked to the application under test, as well as to the ATTOL runtime. The test execution produces a trace file, which is analyzed by the ATTOL postprocessor to issue a *test report*. While ATTOL is usually run via its GUI, it can also be controlled in a non interactive way via command lines.

The coupling with ATTOL did not require any modification of the SESAME core modules. It only impacted the external Shell scripts called by the mutant runner. The corresponding effort was no greater than the usual effort to tune SESAME to new experimental settings. Specifically, the Shell scripts were modified as follows:

- The *preprocessing of a test set* (here, a single test *plan* may be considered) uses ATTOL to perform a golden run with the original source program. The ATTOL test report is then filtered to remove timestamp information (date of the report, etc.). This is intended to facilitate comparison with the mutants: a mutant is not killed if the filtered test reports are identical.
- For each mutant, the *compilation/link program* uses the ATTOL preprocessor to build the executable associating the mutant with the test plan. SESAME's standard *mutant execution program* can then be used.
- The *postprocessing of a test set* uses the ATTOL postprocessor to analyze the mutant trace. The resulting report is then filtered before comparison with the golden report.
- As the duration of a session may be long, the *closing session program* has been enriched by a small utility sending an E-mail to the operator to signal the end of the experiments.

As can be seen, no specific difficulties arose during the interfacing exercise.

3. Dependability research using SESAME

SESAME is potentially useful whenever there is a need to perform some experimental assessment in the presence of software development faults. However, the use of mutations to emulate development faults raises the controversial issue of the representativeness of this fault model with respect to real faults. This issue was the motivation for an empirical study involving both real faults and SESAME-generated mutations, which supplied positive results summarized in Section 3.1. The two following sections report on work addressing fault removal and fault tolerance issues. Section 3.2 reports on work that investigated a probabilistic testing approach for fault removal. With SESAME support, the fault revealing power of the so-called *statistical testing* approach was assessed and compared to that of other approaches. Section 3.3 introduces on-going work related to fault tolerance in autonomous robots. SESAME is used to inject development faults into the decision functions of the robot, with the aim of assessing the impact of the faults in the absence and then the presence of fault tolerance mechanisms.

3.1. Mutations as a model for emulating software development faults

The representativeness of the fault model is a recurring concern in the framework of fault injection. Specifically, the ability of mutations to emulate actual development faults was investigated by our colleagues Daran and Thévenod-Fosse in [7].

At the time of this study, some encouraging results were already available from the testing community. There was empirical evidence that test data generated to reveal usual (1st order) mutations could reveal more complex mutations [18] or even complex known programming faults [10]. Also, [23] observed that mutations could produce subtle erroneous behaviors like: (i) turning a combinational function into a sequential one, or (ii) producing intermittent failures under unforeseeable conditions[2]. This yielded Daran and Thévenod-Fosse to conjecture that the errors (incorrect internal states) and failures (incorrect output results) produced by mutations could be similar to those produced by real faults. Taking the example of a C program of about 1K lines of code, they performed a detailed analysis of the errors produced by 12 real faults and 24 mutations generated by SESAME. The focus was on the mechanisms of run-time error creation, masking and propagation up to failure occurrence. Overall, the results involved 3730 recorded

[2] In the dependability literature, such faults whose activation pattern is not systematically reproducible are called *elusive* faults [4]. It is well-known that most residual development faults in large software systems are elusive.

errors (1458 errors produced by real faults and 2272 by mutations). They supported a good representativeness of mutations in terms of the errors produced:

- 85% of the errors produced by the mutations were also produced by the real faults.
- At a finer level of analysis, the error flow graphs were also found to be quite similar to those induced by real faults. The similarities manifested in identical flows after a possibly different root error (which depends on the specific fault activated), or in common subflows resulting in the same effect on output variables (with error cancellation, masking or failure). Complex error behaviors were observed.
- From a testing perspective, the consequence is that it may be as difficult to reveal a mutation as to reveal a real fault.

Hence, while mutations are not syntactically close to real faults (whose correction usually impacts several instructions, possibly spread over several instruction blocks), it may be relevant to use them as a fault model as long as the experimental assessment depends on the representativeness of the *errors*. In particular, mutation analysis should be adequate to assess the revealing power of test sets.

3.2. Assessment of statistical testing for fault removal

Statistical testing [23] is a probabilistic approach for test generation that has been developed at LAAS during the nineties. It aims to compensate for the imperfect connection of common test selection criteria with the faults to be revealed: the cases identified by a criterion have to be exercised several times with different random data. In this way, there is no need for a perfect match between identified test cases and revealing inputs. Statistical testing should not be confused with operational testing, or with (blind) random testing. Since the sampling profile is determined based on a test criterion, it may have little connection with actual usage. Also, it is usually very different from a uniform profile over the input domain.

SESAME has supported ten years of experimentation on statistical testing. Most of our case studies were critical software developed by industrial partners from the avionic, space and (civil and military) nuclear domains. Very few (if any) real fault reports were available to us, and in most cases the software had already been subjected to a thorough validation process. Hence, the experimented test methods could not be assessed based on real faults alone, so it was decided to use mutation analysis. The target programming languages were as diverse as Assembler, C, Pascal and LUSTRE. The size of the source code ranged from about 100 lines of code to thousands.

Table 3 provides an overview of the experiments performed with SESAME. Most of them are reported in [24] with appropriate pointers to the original papers. Experiments were performed on different categories of test sets:

- Statistical test sets designed from structural or functional criteria.
- Deterministic test sets designed from structural criteria (test sets were manually selected to cover the target criterion).
- Random test sets generated according to a uniform profile over the input domain.
- Industrial test sets (deterministic).

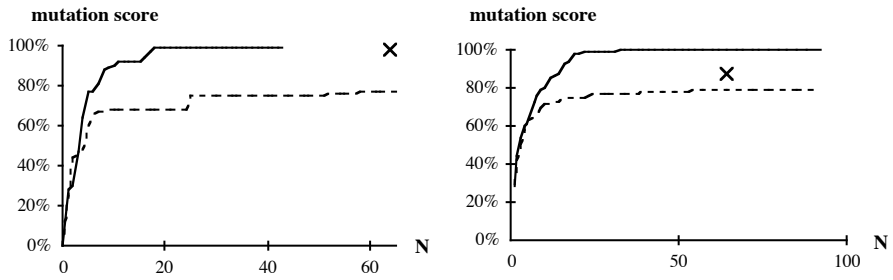
The assessment involved the measurement of the mutation score, the identification of the subsets of mutants that were killed or not by the test sets, or the evolution of the mutation score as a function of the test size (for statistical and random test sets).

Language	# prog.	# mutants	# test sets
C	11	26,324	173
Lustre	8	5,119	60
Pascal	2	249	22
Assembler	4	1,839	34

Table 3. Overview of SESAME case studies

We provide below some conclusions that were drawn from the experiments:

- Random uniform testing turned out to be a poor strategy, except for some very simple software components. The analysis of the evolution of the mutation scores showed that the inadequacy of the uniform profiles could not be compensated by any reasonable increase in the test sizes.
- There was no empirical evidence that deterministic structural test sets were any more effective than purely random ones, and this whatever the stringency of the adopted criterion. This confirms the tricky link between these criteria and the faults they aim to track down.
- On the contrary, high mutation scores were repeatedly observed for statistical testing. The approach allowed us to increase significantly the failure probability of programs, even with respect to faults loosely connected with the criteria. To illustrate the fault revealing power of statistical testing, Table 4 shows results involving both real faults and mutations. The target programs are an industrial program (IND) and a student version (STUD) developed from the same specification. The statistical test sets (ST-Sets), each containing 441 inputs, are designed from a weak functional criterion for Statechart models (the coverage of basic states). Five ST-Sets have been generated to



(a) Prog1, 118 mutations (b) Prog2, 6150 mutations

Figure 7. Comparison with industrial test sets

study the reproducibility of results. One large uniform test set (U-Set) is also provided for the purpose of comparison.

- The comparison with industrial test sets, which could be performed for some software units coming from different industrial companies, showed the higher efficiency of statistical testing. In some cases, the industrial test sets were as effective as the statistical ones, but longer (Fig. 7.a). In the other cases, the industrial test sets were shorter but supplied a lower score than the statistical test sets truncated to the same length (Fig. 7.b).

Today, statistical testing is no longer a research object for us. However, it is routinely used in the framework of our testing research. For example, [16] has investigated the design of tests starting from partial proofs. The proposed approach makes use of a probabilistic generation of test data. It is also worth noting that other authors have investigated a different implementation of statistical testing [11] based on the generation of combinatorial structures and on randomized constraint solving. To compare their approach to "standard" statistical testing, they studied one of the C programs included in Table 3 and used the SESAME database of mutants for this program.

	Real faults		**Mutation scores**	
	IND 1 fault	STUD 12 faults	IND 3756 mut.	STUD 2419 mut.
ST-Sets (N=441)	Always revealed	Always revealed	[0.93, 0.961]	[0.998, 1.0]
U-Set (N=5300)	Not Revealed	5 revealed	0.582	0.743

Table 4. Revealing power of statistical test sets

3.3. Assessment of fault-tolerance in robot planning

SESAME is currently used to provide fault injection in the SAC[3] project [17], which is investigating techniques for tolerating development faults in the decision mechanisms of critical autonomous robot systems.

The focus is on fault tolerance techniques aiming at improving reliability of planning mechanisms through diversification of the models on which planning is based. The efficiency and the performance of these fault tolerance techniques is being evaluated through fault injection into real robot software executed in a simulation environment. The targeted system is an autonomous rover whose mission includes taking photographs at several geographic locations, communicating with an orbiter during pre-determined visibility windows, and returning to its starting point within a fixed deadline. Fault injection is achieved by mutation of the planning model.

3.3.1. Particularities regarding autonomy

Several aspects of autonomous systems complicate the tasks of mutant execution and comparison.

- The asynchrony of the various subsystems of the robot and the underlying operating system causes nondeterminism in the experiments: task scheduling differences between similar experiments may degrade into task failures and possibly unsatisfied goals, even in the absence of faults. Several equivalent experiments must thus be carried out in order to obtain a statistically representative result.
- A particularity of autonomous systems is their ability to adapt to different situations by virtue of the solution space search embodied in their decision mechanisms. Thorough testing of decision mechanisms must therefore confront them with a wide variety of situations. In the SAC experimental

[3] French acronym for "Critical Autonomous Systems".

framework, this is addressed by considering five different missions (defined by different sets of objectives) and four different environments (defined by the number and the location of obstacles with which the rover may be confronted), leading thus to 20 different execution contexts for each mutation.

- Contrary to more classic mutation experiments, the result of an experiment cannot be easily dichomotized as either "failed" or "successful". Indeed, an autonomous system is confronted to partially unknown environments and situations, and since some of its objectives may be difficult or even impossible to achieve in a particular execution context, the assessment of the results of a mission must be graded into more than two levels. In the SAC framework, the quality of the result of a given experiment incorporates first the subset of goals that have been successfully achieved, and second, performance results such as the mission execution time and the distance covered by the robot to achieve its goals.
- The problem of equivalent mutants is exacerbated in the non-deterministic context of autonomous systems: even if two mutations lead to different subsets of achieved goals or dissimilar performance results, they might nevertheless be equivalent. Detecting "true" equivalence is therefore not a simple matter.

3.3.2. Implementation

Due to the impossibility of dichotomizing the results of an experiment as "success" or "failure", the SESAME tool has not been used to execute the mutants, but solely to inject faults in the planning model. This model is written in the specific declarative language of the IxTeT planner [12] considered in the SAC project. It is composed of tasks described by constraint relations on numerical and temporal variables, and assertions (*hold* or *event*) on the system attributes, which are the different variables that together describe the system state. Figure 8 gives an example of a task described in the IxTeT language.

```
task TAKE_PICTURE (?obj, ?x, ?y)(t_start, t_end){
  ?obj in OBJECTS;
  ?x in ]-oo,+oo[; ?y in ]-oo,+oo[;

  hold(AT_ROBOT_X():?x,(t_start,t_end));
  hold(AT_ROBOT_Y():?y,(t_start,t_end));
  hold(PTU_POSITION():downward,(t_start,t_end));

  event(PICTURE(?obj,?x,?y):(none,doing),t_start);
  hold(PICTURE(?obj,?x,?y):doing,(t_start, t_end));
  event(PICTURE(?obj,?x,?y):(doing,done),t_end);

  (t_end - t_start) in ]0,60];
}nonPreemptive
```

Figure 8. Example of IxTeT task

The SESAME mutation tables were written after a manual survey of the model syntax aimed at (a) grouping similar and exchangeable elements, and (b) reducing the size of the tables and the effort needed to produce them. An extract of the mutation tables can be found in Figure 9. It includes four main types of substitution:

- Substitution of numerical values: each numerical value is exchanged with members of a set of real numbers that encompasses (a) all numerical variables in all the tasks of the model, (b) a set of specific values (such as 0, 1 or -1), and (c) a set of randomly-selected values.
- Substitution of variables: since the scope of a variable is limited to the task where it is defined, numerical (resp. temporal) variables are exchanged with all numerical (resp. temporal) variables of the same task.
- Substitution of attribute values; attribute values are exchanged with other possible values in the range of the attribute. Due to the simplicity of SESAME, a problem may arise when the ranges of different attributes overlap since a value corresponding to one attribute may be replaced by a value from the range of a different attribute. However, such mutants generate errors during compilation and are thus discarded by SESAME before execution of the experiments.
- Substitution of language operators: in addition to classic numerical operators on temporal and numerical values, the IxTeT language employs specific operators, such as "nonPreemptive" that indicates that the task being described cannot be interrupted by the executive.

```
substitution of numerical values
«-oo» <---> «+oo», «0», «60», «1», «-1»,
            «-4», «26.3»

substitution of numerical and
temporal variables
«?obj» <---> «?x», «?y»
«t_start» <---> «t_end»

substitution of attribute values
«downward» <---> «straigth», «other»
«none» <---> «done», «doing»

substitution of operators
«nonPreemptive» <---> «latePreemptive», «»
«+» <---> «-», «*», «/»
```

Figure 9. Example of SESAME mutation table

Two other types of mutation have been implemented, although not with the SESAME tool: first the removal of a randomly selected constraint relation in the planning model, second the addition of a syntactically correct line, generated randomly from a lexicon of the planning model.

Our preliminary mutation experiments are aimed at evaluating the baseline resilience of a non-redundant planner. The first results show a wide variety of execution behaviors, from failure of the planner (crash, hang, or timeout), to equivalent executions. Typically, equivalent executions (and thus equivalent mutants) can be found when the mutation relaxes a numeric or temporal constraint, which may have no incidence on the plan because of the intrinsic resilience of constraint planning: the constraint may also be enforced in another part of the model, or the modification may not have sufficient impact to cause a plan failure. Dissimilar executions include failed objectives (e.g., due to a faulty initialization, or a wrong execution mode for a task), bad performance (e.g., due to a wrong estimation of the distance between two locations), or more complicated behavior, such as an increase in system performance at the cost of some failed objectives (e.g., caused by the impossibility to abort movement tasks).

4. Conclusion

The design of SESAME was driven by a pragmatic concern: to offer flexible support to experimental research. This proved a successful approach, since the tool has been used for fifteen years to tackle different targets, with sometimes quite different research objectives. During this period, the maintenance of the tool was also facilitated by its design. The evolutions mainly concerned the enrichment of the mutation tables and of postprocessing utilities, which could easily be added without affecting the core modules. Also, the Pascal code of the early versions has been automatically translated into C, to improve portability.

Transfer to industry was less successful. SESAME was transferred to two partners, but it seems that the experience did not go beyond R&D studies. While the partners were convinced of the relevance of mutation analysis, the manual detection of equivalent mutants proved prohibitive. SESAME leaves open the possibility of using external tools for the detection of equivalence, but currently very few such tools are available. For Fortran targets, the Equalizer module of Mothra uses heuristics derived from compiler optimization techniques [19]. The tool was experimentally found to detect about 10% of the equivalent mutants on 15 small programs, which is quite helpful but far from sufficient from an industrial viewpoint. More powerful detection approaches based on constraint solving [21] or program slicing [13] are, to our knowledge, at best implemented in proof-of-concept prototypes.

For our current work on fault-tolerance in robot planning, the detection of equivalent mutants is specifically difficult, because of the peculiarities of the target language and due to execution non-determinism. Fortunately, other problems related to mutation analysis are less specific, and may be addressed by adapting existing solutions. For example, we cannot afford a large number of mutants because each individual mutant has to be run 60 times (there are 20 execution contexts, and 3 runs per context to study non-determinism). To address this problem, we are planning to use the principle of selective mutation [20] combined with random sampling over the set of other mutants. Determining an adequate selection strategy for our target will of course require experimentation.

As regards the computational cost, we experienced that the cost incurred by the separate compilation of each mutant is negligible compared to the cost of the experimental runs. We are currently investigating the possibility of developing grid-based support for the execution of SESAME mutants. In the context of our current experiments on robot planning, this implies a non-trivial porting exercise since the mutant execution environment includes the whole robot software and a simulator of its environment.

5. Acknowledgements

SESAME received financial support from the *Région Midi-Pyrénées*. It received the innovation prize from the *Association pour le Développement de l'Enseignement, de l'Économie et des Recherches de Midi-Pyrénées*. The coupling with ATTOL was funded in part by Technicatome.

We would also like to mention the significant contributions to the development and use of SESAME of Pascale Thévenod-Fosse, Muriel Daran and Christine Mazuet.

6. References

[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J-C. Fabre, J-C. Laprie, E. Martins and D. Powell, "Fault injection for dependability validation: a methodology and some applications", *IEEE Trans. on Software Engineering*, Vol.16, no.2, pp.166-182, 1990.

[2] J. Arlat, J. Boué, Y. Crouzet, E. Jenn, J. Aidemark, P. Folkesson, J. Karlsson, J. Ohlsson and M. Rimen, "MEFISTO: a series of prototype tools for fault injection into VHDL models", in *Fault injection techniques and tools for embedded systems reliability evaluation*, Kluwer Academic Publishers, ISBN 1-4020-7589-8, pp.177-193, 2003.

[3] J. Arlat, J-C. Fabre, M. Rodriguez and F. Salles, "MAFALDA: a series of prototype tools for the assessment of real time COTS microkernel-based systems", in *Fault injection techniques and tools for embedded systems reliability evaluation*, Kluwer Academic Publishers, ISBN 1-4020-7589-8, pp.141-156, 2003.

[4] A. Avizienis , J-C. Laprie, B. Randell and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, pp.11-33, 2004.

[5] P. Chevalley and P. Thévenod-Fosse, "A mutation analysis tool for Java programs", *Int. Journal on Software Tools for Technology Transfer (STTT)*, vol. 5, no. 1, pp.90-103, 2003.

[6] Y. Crouzet, P. Thévenod-Fosse and H. Waeselynck, "Validation du test du logiciel par injection de fautes: l'outil SESAME", in *11ème Colloque National de Fiabilité et Maintenabilité*, Arcachon, France, pp.551-559, 1998.

[7] M. Daran and P. Thévenod-Fosse, "Software error analysis: a real case study involving real faults and mutations", *International Symposium on Software Testing and Analysis (ISSTA'96)*, San Diego, USA, ACM Press, pp.158-171, 1996.

[8] M. Delamaro and J. Maldonado, "Proteum – A Tool for the Assessment of Test Adequacy for C Programs", *Conference on Performability in Computing Systems (PCS 96)*, New Brunswick, USA, pp. 79-95, 1996.

[9] R.A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer", *IEEE Computer*, vol. 11, no. 4, pp. 34-41, 1978.

[10] R.A. DeMillo and A.P. Mathur, "On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software", Software Engineering Research Center Report, Purdue University, W. Lafayette, USA, 1994.

[11] A. Denise, M.-C. Gaudel and S-D. Gouraud, "A Generic Method for Statistical Testing", *15th IEEE International Symposium on Software Reliability Engineering (ISSRE'2004)*, Saint-Malo, France, IEEE CS Press, pp.25-34, 2004

[12] M. Ghallab and H. Laruelle, "Representation and Control in IxTeT, a Temporal Planner", *2nd Int. Conf. on Artificial Intelligence Planning Systems (AIPS-94)*, Chicago, IL, USA, pp.61-67, AIAA Press, 1994.

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

[14] K. Kanoun, Y. Crouzet, A. Kalakech, A. E. Rugina and P. Rumeau, "Benchmarking the Dependability of Windows and Linux using Postmark Workloads", *16th Int. Symp. on Software Reliability Engineering (ISSRE 2005)*, Chicago, IL, USA, IEEE CS Press, pp.11-20, 2005.

[15] K. King and J. Offutt, "A Fortran Language System for Mutation-based Software Testing", *Software Practice and Experience*, vol. 21, no. 7, pp. 685-718, 1991.

[16] G. Lussier and H. Waeselynck, "Deriving test sets from partial proofs ", *15th IEEE International Symposium on Software Reliability Engineering (ISSRE'2004)*, Saint-Malo, France, IEEE CS Press, pp. 14-24, 2004.

[17] B. Lussier, A. Lampe, R. Chatila, J. Guiochet, F. Ingrand, M.-O. Killijian and D. Powell, "Fault Tolerance in Autonomous Systems: How and How Much?" in *4th IARP - IEEE/RAS - EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*, Nagoya, Japan, 2005.

[18] J. Offutt, "The coupling effect: fact or fiction?", *3rd Symposium on Testing, Analysis and Verification (TAV 3)*, Key West, USA, pp. 131-140, 1989.

[19] J. Offutt and M. Craft "Using Compiler Optimization Techniques to Detect Equivalent Mutants", *Journal of Software Testing, Verification, and Reliability*, vol.4, no. 3, pp.131-154, 1994.

[20] J. Offutt, A. Lee, G. Rothermel, R. H. Untch and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators", *ACM Trans. on Software Engineering Methodology*, vol.5, no. 2, pp. 99-118, 1996.

[21] J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths", *Journal of Software Testing, Verification, and Reliability*, vol 7, no. 3, pp.165-192, 1997.

[22] J. Offutt and R. Untch, "Mutation 2000: Uniting the Orthogonal", in *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, pp. 45-55, 2000.

[23] P. Thévenod-Fosse, H. Waeselynck and Y. Crouzet, "Software statistical testing", in *Predictably Dependable Computing Systems,* B. Randell, J-C. Laprie, H. Kopetz & B. Littlewood (Eds), Springer Verlag, pp. 253-272, 1995.

[24] P. Thévenod-Fosse and H. Waeselynck, "Software statistical testing based on structural and functional criteria", *11th International Software Quality Week (QW'98)*, vol. II, San Francisco, USA, 1998