Université Catholique de Louvain
Louvain School of Engineering
Computing Science Engineering Department

Modeling operation errors by model mutation

Supervisors: Pr. Charles Pecheur
Sébastien Combéfis
Reader: Bernard Lambeau

Master thesis presented in partial fulfillment of the requirements for the degree of:
Master of science in Computer Science and Engineering
with an option in:
Security and Networking
by:
Olivier Goletti

Louvain-la-Neuve
Belgium June 2011

ABSTRACT

This master thesis discusses the application of mutation-based testing to human-machine interactions (HMI) analysis. The objective is to be able to evaluate the robustness of a mental model, which represents the way a user sees a specific system with respect to a system model which is the formal representation of the internal states and transitions of a given system. The main idea is to consider human cognitive imperfections leading to operation errors and model them as mutations in this mental model. A set of mutation operators is provided in this document and is used in a proposed framework. This tool uses to produce mutants and to evaluate them against properties which characterize the control the user has on the system. The results of such tests leads to the actual evaluation of the robustness of the evaluated mental model. This document gathers theoretical material on both the HMI and the mutation-based testing fields. It also provides experimental results of the process on the model of a vehicle transmission system and on a model of the Therac-25 medical device. Those results have shown that the proposed framework is able to evaluate the robustness of a mental model.

I wanted to personally thank some people who helped me going through this huge task which is a master thesis.
Thanks to Pr. Pecheur, for your relevant comments, your good advice and your support.
Thanks to Sébastien Combéfis, for your rereadings, your availability and your shared knowledge in the field, in Java, in LaTeX, etc.
Thanks to Bernard Lambeau for agreeing to be a reader of this master thesis.
Thanks to my roommates and friends for the good mood and the basketball.
Thanks to my parents and my family for your unswerving support.
Thanks to my girlfriend for the comprehension, for being there, for everything...

TABLE OF CONTENTS

Abstract **iii**

Acknowledgments **v**

Table of contents **vii**

Introduction **1**

1 Human-machine interactions **3**
1.1 Introduction of the concept 3
1.2 Labelled transition system as modeling tool 4

2 Mutation testing **7**
2.1 Introduction of the concept 7
2.2 Specification mutation 10

3 Methodology **13**
3.1 Process design 13
3.2 Mutation operators 14
3.3 Properties 16
3.3.1 Full-control 16
3.3.2 Alternative full-control 17
3.3.3 Reachability 18
3.4 Mental model generation 18
3.4.1 A minimization-based approach 18
3.4.2 A learning approach 18

4 Architecture and implementation **21**
4.1 The `lts` package 21
4.2 The `mut` package 28
4.3 The `checkProp` package 32
4.4 Usage 34

5 Experiments **37**
5.1 Tests on the gearbox model . . . 37
5.1.1 The model . . . 37
5.1.2 Preliminary results . . . 38
5.2 Case study: the Therac-25 . . . 44
5.2.1 Presentation of Therac-25 . . . 44
5.2.2 Different mental models . . . 45
5.2.3 Results . . . 46
5.2.4 `checkProp` analysis . . . 47
5.3 Analysis . . . 48
5.3.1 About the mutants . . . 48
5.3.2 About the properties . . . 48
5.3.3 About the robustness . . . 49

Conclusion **51**

Bibliography **53**

Appendices **59**

A API of the `lts` package **59**
A.1 Interfaces . . . 59
A.1.1 `LabelIF` . . . 59
A.1.2 `StateIF` . . . 59
A.1.3 `TransitionIF` . . . 60
A.2 Abstract classes . . . 60
A.2.1 `StateAbstract` . . . 60
A.2.2 `TransitionAbstract` . . . 60
A.3 Classes . . . 61
A.3.1 `LabelImpl` . . . 61
A.3.2 `StateImpl` . . . 61
A.3.3 `CompState` . . . 62
A.3.4 `TransitionImpl` . . . 63
A.3.5 `LTS` . . . 63
A.3.6 `LTSLoader` . . . 67
A.4 Inner classes of `LTS` . . . 67
A.4.1 `InnerState` . . . 67
A.4.2 `InnerState` . . . 68

B API of the `mut` package **71**
B.1 Interface . . . 71
B.1.1 `MutOpIf` . . . 71

B.2 Abstract classes 71
B.2.1 `MutOpAbstract` 71
B.3 Classes 72
B.3.1 `COL` 72
B.3.2 `Pair` 73

C API of the `checkProp` package **75**
C.1 Classes 75
C.1.1 `Checker` 75
C.1.2 `StatTableEntry` 76

INTRODUCTION

The interactions between a user and a system are studied in the scope of the *human-machine interaction* (HMI) research field. The main goal of HMI analysis is to provide to the user an intuitive and easy way to control a specific system, to interact with it. Therefore, this analysis attempts to avoid situations in which the system behaviour is different from what is expected by the user regarding the information he has about the current state of the system. Indeed, such problems could lead to confusions, misuse of the system and maybe critical failures. The study of these interactions is done with the use of formal methods on models which represent both the user and the system.

Mutation-based testing is a part of the fault-based testing techniques. The main idea behind it is to seed some faults in a program, then to use the modified programs, called *mutants*, to evaluate the ability of a test set to detect the faulty ones. With mutation testing, these altered programs are generated by *mutation operators* which represents common mistakes made by programmers. The different test cases in the test set are then executed with the mutants. A score can then be computed on the effectiveness of the tests to detect, to *kill* the faulty mutants. A higher score is better and so the idea is to improve the test set in order to detect a maximum of mutants. The improvement of the test suite indirectly improves the tested program since faults can be detected.

In the scope of this thesis, the idea was to apply mutation-based testing with HMI analysis. Mutations have been seeded in a formal model, called the *mental model* which represents the user of a system. The mutation operators used represent the most common errors a user can make while using a given system. The application of mutation testing to HMI analysis is to slightly modify a given mental model by generating mutants based on this set of operators. While mutating this model, some properties will be analyzed on both the original mental model and its mutated versions. The goal is to evaluate the robustness of this model against the mutation operators, which represent in this case knowledge errors of the user. This robustness is higher if more mutants succeed the property checks. In other words, it is the opposite of the basic mutation testing since what is wanted now is that mutants pass the checks instead of being killed.

The approach was firstly to chose the right set of mutation operators since they have to represent operation errors of the user. For example, if the user of a specific system forgets to pull a lever or does not hear an alarm signal, the corresponding mutation operator will have to model it. This modeling is done by modifying the mental model. Once they were chosen, the technique was applied on a simple but non-trivial system representing a vehicle transmission system. This example is taken from [HD07]. This step was done to get a first validation of the framework developed and to identify necessary adjustments. Then it was applied to a larger system as case study. The system used is a Therac-25 model

presented in [BBS08]. The Therac-25 is a well-known medical device at the origin of some accidents (which have sometimes led to death) and has been widely studied.

The properties analyzed on the mutants study the control that a user has on the system through those mutated version of the mental model. They focus on whether the user knows at any times his options, the actions he can perform, the observations he should wait for, etc. For the case study, we also looked at whether the mutants cause death or not with a reachability property.

All those features: the modeling, the generation of mutants and the property checks have been implemented in a Java tool: `mentalRate`. This tool is proposed as a framework providing the basis to analyze the robustness of mental model through the use of a mutation-based technique. This framework is also easily extendable.

The main results of this master thesis are interesting and open doors for further study of this technique. Firstly, a set of mutation operators has been composed as a starting point for future enhancements. Secondly, a set of properties, well-known and new ones, has been established and studied. Finally, a process has been designed to analyze a mental model, from its conception to the evaluation of its robustness. This process has been tested with a case study and the results have shown that, as expected, more redundant models are the most robust ones.

This master thesis is divided in five chapters, the first one introduces the theoretical concepts behind HMI and explains the modeling tools used. Chapter 2 gives all the background about mutation testing. The third chapter provides the methodology used in order to actually analyze a mental model, and explains the process designed to do so. Chapter 4 develops the main architectural points of the implementation. Finally, the fifth chapter presents the experiments made. A conclusion on this year's work terminates this document. The complete API of the framework is also available in the appendices A, B and C.

CHAPTER

ONE

HUMAN-MACHINE INTERACTIONS

As explained in the introduction, human-machine interaction is a core part of this master thesis. The aim of this chapter is to introduce the concept of HMI and to present the modelling technique which will be used in this thesis. This chapter will only focus on the formal methods used to study HMI. Of course, there are also other methods which focus on the ergonomic and aesthetic design of interfaces or on the psychological study of such interactions but this kind of studies are out of the scope of this master thesis which focus on behavioral aspects of the HMI analysis.

1.1 Introduction of the concept

HMI studies the interactions between users and machines in general. 'Machines' is a widely used word which stands for automated devices and more precisely in our case for machine-based devices. The main goal of this discipline is a correct and easy use of a machine by a human user called the *operator*. This kind of approach is detailed more precisely in Degani's book: [Deg04].

In fact, when a user is in contact with a machine, the only way for him to interact with it is through the interface. An interface is a limit between two different media which are here the operator and the system. In our case, the interface could either be a physical interface (e.g., mouse, buttons, lever and also gauges, lights, sounds) or a software interface (e.g., field in a form, button on a window, messages, alerts). This kind of interaction is in fact a way to hide the complexity of the underlying behaviour of the machine. The proper functioning of the device is indeed ensured by a lot of internal states and other irrelevant information for the user. In order for a human to be able to use the system, an abstraction of the system is provided, as a user's manual for example. The main goal of this abstraction is to explain clearly the behaviour of the interface without giving irrelevant information about the system.

Getting a good abstraction is a key issue of HMI because since the user does not know the full behaviour of the system, he may misunderstand a signal or he may think the system is in a specific state while it is actually in another one. This could be due to insufficient information on the interface or, in the contrary, due to too much information. This kind of problem may lead, in the case of critical applications, to hazardous situations. For example, the causes of the Kengworth airplane accident [BBB90] were confusing alerts which led the crew to throttle back the wrong engine. This permits to highlight two

main concerns of HMI which are the *correctness* and the *conciseness* of the user's manual. The correctness is defined by Degani as an attribute of an interface with which we "can reliably tell the current state of the machine", with which we "can reliably anticipate the next state of the machine" and which "does not lie to the user". The conciseness aims to make simple interfaces which "are cheaper to produce, make user manuals smaller and less confusing"[Deg04].

1.2 Labelled transition system as modeling tool

This master thesis makes much use of labelled transition systems (LTS) so this section aims to remind the basics about those data structures. The reader familiar with this concept should skip it. For more literature about LTS, see [BG06]. A lot of information has also been found in the first sections of [CP09].

An LTS is an enriched version of a *finite state machine* (FSM). A FSM is a graph composed of a set of states $\mathcal{S}$ (the nodes), an initial state s_0 and a transition function $\mathcal{T}$. The kind of FSM used in this thesis adds the notion of actions from a set of actions $\mathcal{L}$ with which every transition is labelled. This set is called the *alphabet* of the LTS. Therefore, the transition function could be written as a ternary relation: $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$. An element of this set, $(s, \alpha, s' \in \mathcal{T})$ is written as $s \xrightarrow{\alpha} s'$. So it is possible to fully describe a given LTS with the following quadruplet $\langle \mathcal{S}, \mathcal{L}, s_0, \mathcal{T} \rangle$. From this definition, and with σ representing a sequence of actions from the alphabet of the LTS denoted $\sigma = \alpha_1 \alpha_2 \cdots \alpha_n \in \mathcal{L}^*$, an *execution* from an LTS is written $s_0 \xrightarrow{\sigma} s_n$ where σ is called the *trace* of the execution. In the trace of an execution each action α_i is labelling the corresponding ith transition of the execution. The sequence of those transitions is denoted as $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \cdots \xrightarrow{\alpha_n} s_n$. [1]

HMI has been widely studied in the literature through *formal methods* in order to observe, analyze and measure user interactions [DCH00, Rus02, CH03, CHL04, Deg04, HD07, CRB07, CH08, BBS08, TG08, CP09, CPGF]. By using such methods and with the help of mathematical models, it is possible to simulate the parallel functioning of both the user and the system, which provides information about the interaction. A common way to do so, as proposed in [HD07], is to use two LTSs with some enrichments to model those two entities: the model of the machine with all its internal states, events, modes, etc. which is called the *system model* ($\mathcal{M}_M$) and the model of the human user which is an abstraction of the system and is called the *mental model* ($\mathcal{M}_U$).

When LTSs are used for such modeling, the notion of *mode* can enrich them. The modes in a model gather a set of states belonging to a common general behavior of the system and that the user should be able to distinguish. A mode is just an attribute given to each state in order to make a partition of those states and to let the user know in which operating mode is the system.

With this modeling, two types of actions have to be distinguished following the *action-based interface* highlighted in [JP]. The first type is the actions which are controllable by the user. In this set of actions are gathered all the actions which are voluntarily done by the user. Those actions are called *commands*. But there are also actions which are not controllable by the user but that the system is able to do and which are only observable by the user. Those actions are called *observations*. And more formally, for an LTS $\mathcal{M}$, we have that $\mathcal{L}_\mathcal{M} = \mathcal{L}^c_\mathcal{M} \uplus \mathcal{L}^o_\mathcal{M}$, the observable alphabet of an LTS is the union of the alphabet of the commands and the one of the observations [1]. From this distinction, a

[1] Since this master thesis does not use the internal action τ, it is not used in the definitions.

notation describes the set of actions of a certain type available in a specific state of $\mathcal{M}$. This notation is $A^c(s_\mathcal{M})$ for the commands and respectively $A^o(s_\mathcal{M})$ for the observations. Formally, $A^c(s_\mathcal{M}) = \{\alpha \in \mathcal{L}^c | s_\mathcal{M} \xrightarrow{\alpha} s'_\mathcal{M}\}$ and $A^o(s_\mathcal{M}) = \{\alpha \in \mathcal{L}^o | s_\mathcal{M} \xrightarrow{\alpha} s'_\mathcal{M}\}$.

The parallel run of the mental and the system models is done by generating a new LTS which is called the *synchronous parallel composition* of those two models denoted as $\mathcal{M}_M \| \mathcal{M}_U$. In this composition, a state is just a composite state which represents the state in which the system is and the state in which the user is with respect to their models. The initial state of the composition is so a composite state with the initial states of both models denoted as (s_{0_M}, s_{0_U}). Then, a transition with a specific label $(s_M, s_U) \xrightarrow{\alpha} (s'_M, s'_U)$ exists in the composition only if one with the same label exists in the corresponding models from each of the states in the composite state; i.e. $s_M \xrightarrow{\alpha} s'_M \in \mathcal{T}_M \wedge s_M \xrightarrow{\alpha} s'_M \in \mathcal{T}_M$. A new composite state is reached following this transition and it is made of the two destination states in the respective models. One such composite state is created for each pair of destination states reachable from the current pair of states in their respective model. This means that if there is more than one outgoing transition with this label from one of the two states, which means there is *non-determinism*, more than one composite states will be created in the composition.

This modeling technique allows to work on different criteria of the interface and to track errors. Here are three criteria proposed in [HD07]:

- No error state: a state in which the user thinks he is in a specific state while the system is in fact in another one.
- No restricting state: a state in which the user does not have access to a specific transition while it is available in the system.
- No augmenting state: a state in which the user have the information that a transaction is available while it is not allowed in the system.

These are the kind of properties to verify for a correct and coherent mental model. Another set of properties proposed in [CH08] is:

- Feedback: verifies whether a given action provides feedback, it means that the interface is updated each time an action is done in order for the user to be sure that it was taken into account.
- Behavioral consistency: verifies whether a given action causes consistent effect, it means that an action must always cause the same effect in the user interface. It is a generalization from the feedback property.
- Undo: checks whether an action can be undone.
- Reversibility: checks whether an action can be eventually undone

The first set of properties describes properties on the states of a FSM modeling the mental model. Those are behavioural ones, while the second set describes some usability properties. The behavioural properties are interesting ones to study in the scope of this thesis. The first set is the kind of properties on which one can easily work. This kind of properties has been widely studied, see for example approaches using model-checking in [Rus02] and in [CHL04]. Other approaches have been studied: with theorem prover in [CRB07] and in [DCH00], or with graph theory in [TG08].

Those properties are not used in themselves in this work but the ones actually used are related to this behavioural class of properties and are described in more details in Section 3.3.

CHAPTER

TWO

MUTATION TESTING

As mentioned before, the interest of *mutation-based testing* in the scope of this master thesis is to apply mutation analysis on the modeling of HMI in order to test the robustness of a given mental model with respect to a chosen set of properties. This will be done by mutating the mental model. The mutants will be generated with mutant operators representing common operations errors made by a user. These mutants will be executed against the properties to see whether they still respect them or not.

This chapter first introduces the concept of mutation testing and then details the application of this technique to FSM. For more details on mutation testing, the two following survey papers cover very well the topic : [OU01] and [JH10]. The reader may also take a look at the very intuitive introduction of the thesis of Mattias Bybro: [Byb03].

2.1 Introduction of the concept

Fault-based testing is a way to test a program to demonstrate that it does not contain pre-defined faults. For example, to prove that a given program handles division by 0, a test with 0 as divisor will be included. It is important to remember that the goal of a good test suite is not to prove the correctness of a program. It is to highlight not yet uncovered errors in order to correct the program.

Mutation testing is part of this category of software testing. This technique was introduced in 1971 in a paper by Richard Lipton ([Lip71]) which was written following two papers seen by the community as the seminal papers: [DLS78] and [Ham77].

The main principle of this field is namely to mutate a program. It means to slightly change the program in a specific pre-defined way, by inserting one or more errors in it. The goal is to produce alternate programs called *mutants*. Assuming that we have a test suite $\mathcal{T}$ to test the program $\mathcal{P}$, the mutants will help us to rate $\mathcal{T}$ and indirectly to improve $\mathcal{P}$. Indeed, adding a test case in order to kill a mutant which remained alive improve directly the test suite and it is obvious that a better $\mathcal{T}$ contributes to improve $\mathcal{P}$.

The fundamental assumptions of mutation testing are:

- The *Competent Programmer Hypothesis* (CPH) which states that programmers tend to produce programs that are close to the correct program. From the CPH, we assume that only simple likely faults are made and so that only such faults will be inserted into mutants.

- The *Coupling Effect Hypothesis* (CEH) which states that complex faults are linked to simple faults such that if we have a test data set capable to detect the simple faults, it will also detect most of the complex faults.

Those two hypotheses where expressed for the first time in [DLS78]. The CPH was theoretically discussed in [BDLS80] and the CEH was demonstrated in [Wah95] and [Wah00]. They allow mutation testing to only focus on simple likely faults. Mutants containing a single error are called *first-order* mutants and they represent one single error. Those with more than one error are called *high-order* mutants. Thanks to those two hypotheses, mutation testing will only care about first-order mutants which will just be referred as mutants from now.

The generation of mutants is based on a set of *mutation operators* which will represent a predefined set of faults. For example, the well-known set of mutation operators *Mothra* is presented in Table 2.1. These operators are the representation of the most common errors made by programmers in Fortran programs. For example, the AOR operator could be illustrated by the replacement of a $+$ in a program by another arithmetic operator among $\times$, $/$ and $-$.

Type	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	`DO` statement alterations
DSA	`DATA` statement alterations
GLR	`GOTO` label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	`RETURN` statement replacement
SAN	statement analysis (replacement by `TRAP`)
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Table 2.1: The 22 Mothra Fortran mutation operators (adapted from [KO91])

Now let us come back at our program $\mathcal{P}$. $\mathcal{T}$ is a test data set for $\mathcal{P}$, which is assumed to be correct, i.e. passing $\mathcal{T}$. If we now generate mutants of the program thanks to a set of mutation operators some of them will also pass $\mathcal{T}$ while others will fail it. The ones which failed are said to be *killed*, i.e. the error has been detected. The remaining ones are the *alive* mutants. Each mutant which passes $\mathcal{T}$ is a potential error in $\mathcal{P}$ since it's an error which is not detected by $\mathcal{T}$. Identifying alive mutants can improve the program

by forcing the programmer to add another test case to kill them and, by doing so, maybe to find errors in $\mathcal{P}$. It is also possible to have mutants which will never be killed since they are equivalent versions of $\mathcal{P}$. Such a mutant is called an *equivalent* mutant and is undetectable automatically since program equivalence is an undecidable problem.

We said earlier that mutation testing was a mean to rate $\mathcal{T}$. Indeed, we can compute the *mutation score* (MS) with the formula 2.1. The aim is to maximize this score and a test suite is said to be adequate if $MS = 1$.

$$MS = \frac{\text{number of killed mutants}}{\text{total number of non-equivalent mutants}} \tag{2.1}$$

With this information, we can see the mutation testing process as a feedback loop on the testing of a specific program. A common used schema is provided at Figure 2.1.

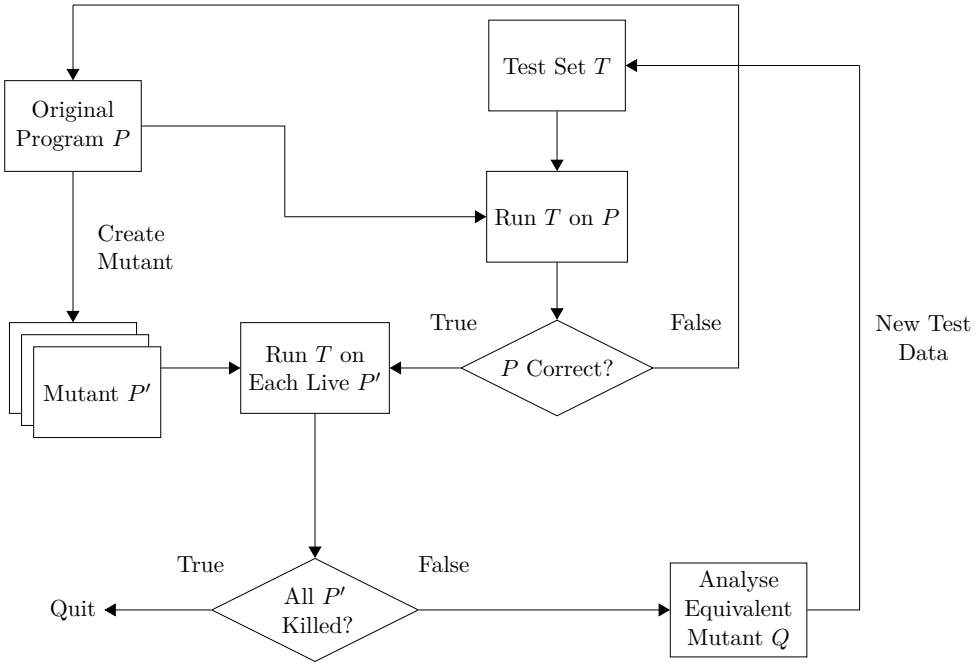

Figure 2.1: Generic Process of Mutation Analysis (adapted from [JH10]).

Since the equivalence problem is undecidable, the programmer may wonder whether the remaining mutants are equivalent or not. This leads to a drawback of mutation testing which, among others, is an impediment to the wide utilization of this technique. The programmer have to check all those remaining mutants to see whether or not they are equivalent to $\mathcal{P}$. Another very time-consuming problem is the generation of all mutants. If one has to exhaustively generate one mutant for each single error at each potential fault place in $\mathcal{P}$, the computational time will raise proportionally. A third difficulty is checking the tests results. Indeed, each time we run a test on a mutant, the programmer has to compare the output with the correct output and decide whether the mutant is killed or not.

All these drawbacks have driven the researchers to develop some optimizations. The most important are:

- *Mutant reduction*: This optimization aims at reducing the number of generated mutants without affecting the mutation score. In order to do so, the idea is to

reduce the number of mutation operators in the operator set. In fact, some mutation operators may produce many more mutants than other ones and many of them could be redundant. For example, ASR and SVR in the Mothra operators are responsible for 40% to 60% of all the generated mutants [KO91]. Reducing the number of mutants could be done by omitting the mutation operators responsible for the largest number of mutants. This technique is called *selective mutation* [ORZ93]. Another way is to choose more "clever" mutation operators which require the most precise test cases. For example, choosing in the Mothra operators set only ABS and ROR because of the cleverness behind the errors. This technique is called *constraint mutation* [WM95]. A technique based on the previous ones is to do a statistical analysis on the set of mutant operators to identify the best subset of mutants which permits to generate less mutants without affecting the mutation score.

- *Execution reduction*: Here the optimization is no more to reduce the number of mutants but to optimize the mutant execution process. In particular, the approach here is the improvement of the decision process which checks whether a mutant is killed or not. *Strong mutation* is the standard way, i.e. the mutant is killed if its output is different from the original program. With *weak mutation*, the mutant is directly checked after the execution of the mutated part of the original program. It reduces indeed the time needed to check all mutants since a complete execution is no more needed for each mutant. It has been shown that in most the common cases, weak mutation is a viable alternative to strong mutation [OL91].

For more details about mutant reduction and execution reduction, the survey paper [JH10] gives a lot of further readings.

2.2 Specification mutation

In this subsection, we will see how the mutation analysis can be transposed to specifications since the application field for this thesis is to apply mutation analysis to FSM which are a way of representing specifications.

Mutation testing has now been widely studied and has been applied both to program and to program specification. We have described it as white-box testing, i.e. we looked into the source code of a program in order to insert faults. But specification mutation belongs to black-box testing. In this case, the faults are seeded into the design level of a program and the testing is oriented to the software function. Again, refer to the survey [JH10] for pointers about this field of application of mutation based testing.

This technique can be applied to any formal language which describes software in abstract terms, these languages are also called model-based languages. The mutating syntax for such an application is very close to the one used for program mutation. One must define a set of mutation operators and apply it on the model. But in the case of model-based mutation, test cases are this time traces which are allowed in the original machine. If this trace is not allowed in a specific mutant, then it is killed.

Killing a mutant can be done automatically through the use of a *model checker*. This kind of tool aims at verifying a property on a *concurrent system*, meaning the execution of different components together and in our case this is the mental and the system model. The main way to express those properties is in *linear temporal logic* (LTL). The temporal logic is able to deal with the time ordering in which events should happen without explicitly

introducing time [CGP]. But LTL is not the only way to do model checking, it is also possible to do it through model comparisons, which is one of the technique we use. The idea is to compare two models by exploring their different states through a traversal and defining an equivalence with respect to a property. Another technique used is *reachability*, which allows to checks whether a given state is accessible. The properties actually used are defined in Section 3.3. And so, using this model checking, we are able to automatically check a desired property on a mutant and to provide an error trace if the mutant is killed.

A good point in the model-based mutation is that the equivalence problem disappears. In fact, this problem becomes decidable since we deal here with finite models. This leads to the fact that we know whether a mutant is equivalent or not directly by using a model checker ([OAL07]).

Mutation testing on FSM has been applied for the first time by Fabbri et al. in [FDMM02]. They proposed a set of mutation operators for mutation analysis on FSM which is adapted and explained here:

arc-missing: a transition is removed from the graph representing the FSM.

wrong-starting-state: the initial state of the FSM is changed.

event-missing: a label of a specific action is removed.

event-exchanged: a label has been exchanged by another one from $\mathcal{L}$.

event-extra: a new action from $\mathcal{L}$ is added to label a transition.

state-extra: a new state is added in the FSM.

Another set of mutation operators could be found in [LDL09] and is adapted here:

Reverse of Transition (ROT): the direction of a transition is reversed.

Missing of Transition (MOT): a transition is lost, or it is not defined.

Redundancy of Transition (DOT): more transitions than expected occur.

Change of Input (COI): the action labelling a transition has been changed.

Missing of Input (MOI): a transition without an action or with a null action.

Start State Changed (SSC): the initial state has been changed.

Start State Redundant (SSR): a new state is introduced in the FSM. It could be redundant.

As explained before, a single operator provides more than one mutant. In this case, except of DOT and SSR which are not quantifiable, all the other mutation operators produce mutants in a polynomial way [LDL09].

The authors who have proposed those set have shown that mutation testing can be effectively applied to FSM on basis of those operators. Fabbri et al. also raised the fact that some specific classes of mutants of order 2 seemed to be very effective for FSM.

The set of operators actually used in this thesis will be discussed in the following chapter and is based on those operators sets.

CHAPTER

THREE

METHODOLOGY

This master thesis is related to HMI because, instead of testing the correctness of the mental model (i.e., the interface), it works on the robustness of such a model. It means that the objective is to see whether the interface is well designed enough to stay coherent with the system even in case of a operation error. This is done through the use of mutation analysis.

In order to do so, a set of errors is defined and some properties, that where verified with the correct mental model, are checked on the mutants to see whether they still remain verified or if not, in which ratio. The errors are modelled as mutations as explained here above. This chapter first presents the process which evaluates the robustness of a given mental model. Then, the different mutation operators and the different properties chosen for the experiments of the thesis are detailed.

3.1 Process design

Figure 3.1 describes the complete process which is used to rate a mental model. This evaluation is the result of an analysis of the robustness of the mental model against the chosen mutation operators and properties. This process has been implemented in a tool called `mentalRate`.

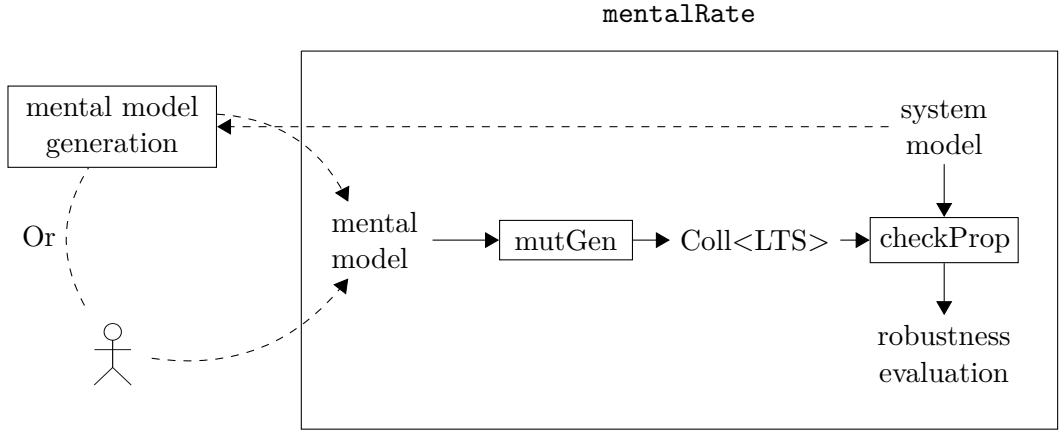

Figure 3.1: From a mental model to a robustness rate

The first input to feed the `mentalRate` process with is a pair of LTSs. One represents a system and the other is a mental model for this system. This mental model can be obtained in different ways. One of them is to manually retrieve it from a user's manual, the other possibility is to generate it based on the system model itself. Several techniques among which a minimization-based approach [CP09] and a learning approach [CPGF] can be used for this problem. Those two techniques are explained in Section 3.4 once the needed properties are defined.

The mental model is passed through the first tool: `mutGen`. This tool is implemented in Java and aims at generating a set of mutants given a mutation operator and an LTS. The number of mutants generated blindly by a specific mutation operator could be very large as explained in the background in Section 2.1. This is the reason why one specific mutation operator is a lazy process which outputs the mutants on demand. Doing so, the user of the application can easily put an upper bound to the number of generated mutants.

The next tool takes as inputs a system as an LTS and a bunch of mutants of a given mental model for this system. The tool provides an output which evaluates the success of the mutated mental models on the system for this property. The different outputs will be detailed in Section 4.4.

3.2 Mutation operators

Based on the lists provided in Section 2.2, this section explains the mutation operators effectively chosen. But before presenting them, the first point to discuss is that those operators have to represent system operation errors. The lists above are mutation operators for generic state machines without particular purpose or meaning in terms of cognitive imperfections. Nevertheless, we have to keep in mind the fact that a mutation operator represents a simple mistake.

The best way to develop rigorously a list of common, basic and simple operation errors is to do a scientific usability study of people using a specific device, recording all the manipulations and then analyzing and statistically populating such a list. A very good discussion comparing usability analysis and formal analysis has been made in [CH03]. But this was not in the scope of this master thesis. So the mutation operators where chosen based on the above lists and on common sense.

Here is the exhaustive list of the mutation operators used. Some visual examples of the different mutants are proposed in Figure 3.2. As a reminder, these mutation operators are defined for LTS:

Change Of Label (COL): the COL mutation operator takes one transition in the LTS and swaps its label with another one from the set of existing labels $\mathcal{L}$. The corresponding manipulation error could be seen as the user mixing up the actions confusing between left and right (Figure 3.2b).

Insertion Of Transition (IOT): in this case, a transition which does not yet exists in the LTS is added between two existing states. The label of this transition is taken from $\mathcal{L}$. In this case, the user just thinks that an additional command or observation is available in a given state (Figure 3.2c).

Missing Of Transition (MOT): the MOT mutation operator simply removes one existing transition from $\mathcal{T}$. It could just be an omission of the user (Figure 3.2d).

MOT2: this operator is an improved version of MOT and aims to be more clever than it in terms of modeling of an operation error. Instead of just removing one transition from s_1 to s_2, it also duplicates the transitions coming to s_1 making them arriving directly to s_2. This is done in order to model the missing of the transition in the mental model in such a way that nothing have to happen to go from s_1 to s_2 (Figure 3.2e).

MOT3: this third version of MOT tries again to be a more realistic one. This time, the mutation operator merges the two states involved in the deletion of the transition. It means the deletion of the two previous states s_1 and s_2 and the creation of a new state $s_1\_s_2$. All transitions going into s_1 or s_2 (except the one has been removed) are redirected towards the new state and all outgoing transitions from s_1 (except the one that has been removed) or s_2 are now going from the new state. This time, the user thinks there is just no difference between the two states (Figure 3.2f).

Reverse Of Transition (ROT): this mutation operator inverts the source and destination states of an existing transition. This could model a mixing up of the user who just swaps the two states (Figure 3.2g).

Starting State Changed (SSC): this mutation just sets another state as the initial state of the LTS. This could model the fact that the user thinks the system should start in a specific configuration while in fact it begins in another one. This could be due to an unclear explanation in the user manual (Figure 3.2h).

Of course, the number of mutants generated by each mutation operator is determined by precise formulas (as a recall, the cardinality of set A is denoted as $|A|$). Those formulas are provided here and the parameters involved are those of the mental model:

COL: $|\mathcal{T}| \times (|\mathcal{L}| - 1)$.

IOT: $|\mathcal{S}|^2 \times |\mathcal{L}| - |\mathcal{T}|$.

MOT: $|\mathcal{T}|$.

MOT2: $|\mathcal{T}|$.

MOT3: $|\mathcal{T}|$.

ROT: $|\mathcal{T}|$.

SSC: $|\mathcal{S}| - 1$.

Those formulas directly lead to the observation that COL and IOT will generate more mutants that the other ones. This is the same kind of reasoning as in Section 2.1 about the optimizations about the mutation operators set. Since the idea here is not to remove mutants, the amount of mutants being generated by COL and IOT has been bounded. Indeed, since the mutation operators produce mutants at random in our framework (Section 4.2) the ratio of succeeding mutant remains significant if the generated amount is sufficiently high. This bound is an adaptation of the idea behind the optimizations which was: the optimization aims at reducing the number of generated mutants without affecting the mutation score.

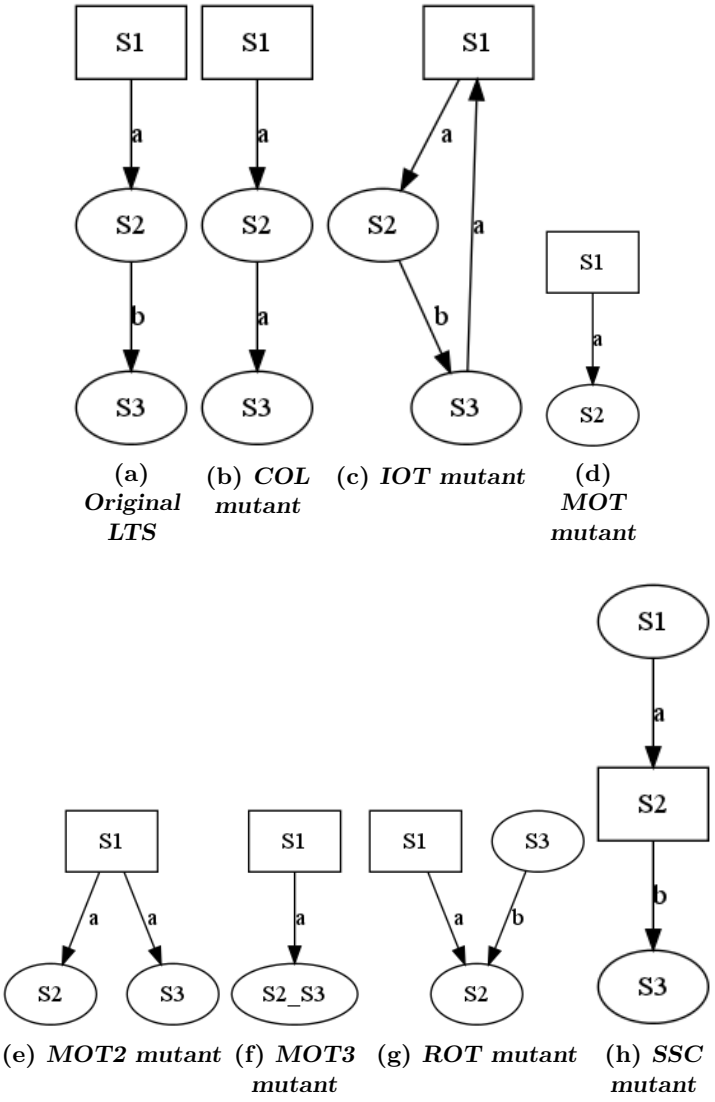

Figure 3.2: Example of mutants resulting of the above list of mutation operators applied on the LTS on Figure 3.2a. The starting state node has a rectangular shape.

3.3 Properties

This section describes in more details the different properties which will be used in the experiments in order to see whether one mutant is killed or not. As a reminder, the reader should keep in mind that the goal here is to see whether a mutant succeeds or not in order to say that the mental model is robust enough to resist to such manipulation errors.

The following properties are also behavioral like the first set in Section 1.2.

3.3.1 Full-control

The first property, and also the strongest one, is the full-control property[CP09]. This property only holds if the user has a total control on the system through its model. Even a small change in the mental model can bring the full-control down because this property

is very strong. We say that a mental model allows full-control of a system if at any time, when using the system according to the mental model, we have that (this is a quotation from [CP09], Section 3.2):

- the commands that the mental model allows are exactly those available on the system;
- the mental model allows at least all the observations that can be produced by the system.

The full-control of a mental model $\mathcal{M}_U$ for a system model $\mathcal{M}_M$ can be formally written as:

$$\forall(s_M \in \mathcal{S}_M,\ s_U \in \mathcal{S}_U,\ \sigma \in \mathcal{L}^{obs*}) \text{ such that } s_{0_M} \xrightarrow{\sigma} s_M \wedge s_{0_U} \xrightarrow{\sigma} s_U,$$

we have:

$$A^c(s_M) = A^c(s_U) \wedge A^o(s_M) \subseteq A^o(s_U)$$

A parallelism can be made with the first set of properties in Section 1.2. Indeed, no augmenting-state corresponds to not having more commands in the mental model and no restricting-state corresponds to not having missing commands or observations in the mental model.

3.3.2 Alternative full-control

While the precedent property is an existing one, this property is new. This property is weaker than the full-control. In fact, it still represents the fact that a user can control the system, but this time, it is not mandatory that the user is able to do any possible commands allowed by the system.

We say that a mental model allows alternative full-control of a system if at any time, when using the system according to the mental model, we have that:

- the system model allows at least all the commands that are available in the the mental;
- the mental model allows at least all the observations that can be produced by the system.

The alternative full-control of a mental model $\mathcal{M}_U$ for a system model $\mathcal{M}_M$ can be formally written as:

$$\forall(s_M \in \mathcal{S}_M,\ s_U \in \mathcal{S}_U,\ \sigma \in \mathcal{L}^{obs*}) \text{ such that } s_{0_M} \xrightarrow{\sigma} s_M \wedge s_{0_U} \xrightarrow{\sigma} s_U,$$

we have:

$$A^c(s_M) \supseteq A^c(s_U) \wedge A^o(s_M) \subseteq A^o(s_U)$$

The same parallelism can be made with the first set of properties in Section 1.2 as above.

3.3.3 Reachability

The last property that is studied is a reachability one. This property is checked by a traversal of the graph to see whether a reachability condition is filled or not.

A state s in an LTS is said to be reachable if there exists an execution leading from the initial state through s. Formally, $s_{\mathcal{M}}$ is reachable in a LTS $\mathcal{M}$ if:

$$\exists \sigma \in \mathcal{L}^{obs*} \text{ such that } s_{0_{\mathcal{M}}} \xrightarrow{\sigma} s_{\mathcal{M}}$$

In the case study, it is applied to see whether a state with some specific properties is reached or not (more details about this is given in Section 5.2.3). The application of this property with the Therac-25 model allows us to checks whether a specific state is reached. In order to check that, a pattern on the name of the state has been identified and the reachability property is verified by doing a traversal of the LTS and verifying in each state whether the name corresponds to the pattern.

3.4 Mental model generation

This section presents the two mental model generation techniques mentioned in Section 3.1, respectively the minimization-based approach [CP09] and the learning approach [CPGF].

3.4.1 A minimization-based approach

This technique is based on the idea of merging the states in the system which can be seen in the mental as having the same behaviour. This merging of states allows to build an abstraction of the system which is the searched mental model. In order to do so, the authors define an equivalence relation based on the full-control property. This equivalence relation states that "two equivalent states must allow the same set of commands but may allow different sets of observations". Then, using this equivalence relation, the minimization is the merging of *fc-equivalent* states. This merge is done in the same way as the merge in MOT3, i.e. redirecting all transitions having one of the fc-equivalent states as destination towards the merged state and all the transitions going from one of the fc-equivalent states are now going from this merged state. The model constituted of those state merged is a mental model for the system, with full-control on the system and minimal in the number of states.

3.4.2 A learning approach

This technique derives a mental model from a given system model and is based on the $L*$ learning algorithm and is explained in [CPGF].

The L* algorithm from [Ang87] aims to learn a language on the basis of words from a given finite alphabet and an oracle saying whether a word is part of not of the language, the *membership query*. The algorithm makes guesses by producing words and build candidates called here *Deterministic Finite Automaton* (DFA). A DFA is a state machine with accepting states which can decides whether a word is part of the language it covers or not. So L*, trough an iteration mechanism, builds candidate DFAs with the oracle responding to membership query. Then, a second type of oracle is needed which answers to *conjectures* and is able to say, given a DFA, if the language covered by this DFA is

or not equal to the searched language and if not, produces a counter-example as a trace. L* is guaranteed to propose conjectures of strictly increasing size until the minimal DFA covering the language is found.

The version in [CPGF] is able to learn a *Three-Valued Deterministic Finite Automaton* (3DFA, notion introduced in [CFC$^+$09]) which will produce a mental model having the full-control on the system. The states of this kind of automaton are partitioned in three types: the accepting, the rejecting and the "don't care" states. A DFA is said to be *consistent* with a 3DFA if and only if all strings accepted by the 3DFA are also accepted by this DFA, and all strings rejected by the 3DFA are also rejected by the consistent DFA. A 3DFA characterize a range of DFAs with the upper bound being the DFA where all the don't care states become accepting ones while the lower bound is the one where they become rejecting ones. The language that the algorithm learns is the range of full-control mental models. This times the membership query is answered with either "yes" if the sequence should be accepted, "no" if it should be rejected or "don't care" in the third case. The conjectures are answered yes if the 3DFA provide full-control and no in the other case. When the minimal 3DFA is found,the corresponding minimal consistent DFA is chosen as the mental model.

CHAPTER

FOUR

ARCHITECTURE AND IMPLEMENTATION

This chapter digs a little bit further in the main architectural aspects of the `mentalRate` tool. The most important features are also developed. It goes through all of the packages constituting the application. The first one is the `lts` package which includes all the implementation of the data structure of an LTS. Secondly is described the `mut` package which contains the implementation of all the mutation operators. The third package is the `checkProp` one in which all the property checks are implemented. Finally, a quick usage section explains how to use the provided application.

4.1 The `lts` package

The implementation of the main data structure of the project was a main concern in this master thesis. It is based on an implementation of Sébastien Combéfis. He makes much use of LTSs in his research work. A class diagram of this package is presented in Figure 4.1.

In this implementation, there are some collections which represents the different parts of the quadruplet presented in Section 1.2: $\langle \mathcal{S}, \mathcal{L}, s_0, \mathcal{T} \rangle$. There is also the principle of an inner representation of the states and transitions. Those internal classes, which names are prefixed with `Inner`, wrap external abstract classes, suffixed with `Abstract` that any user of the LTS type should use. This mechanism is done to hide the structure of the LTS to the outside. In order for the user to interact with the LTS, he should know which external state or transition he wants to manipulate and the links in the structure are hidden, so he can not mess with it. Moreover, it gives the user the possibility to extend the abstract framework proposed without having to change all the internal structure of the LTS. The signature of this class is:

```
1 public final class LTS<S extends StateAbstract, T extends TransitionAbstract> implements Cloneable
```

Each of the following features are presented hereafter. Firstly the abstract framework, then the `LTS` fields, the `Cloneable` implementation and finally the different ways to save a LTS.

This implementation of an LTS also provides a bunch of methods to get information about it and to edit it. The complete list of used functionalities can be found in a detailed specification of the API, in JavaDoc format, is provided in Appendix A (the `lts` package), Appendix B (the `mut` package) and Appendix C (the `checkProp` package). The main

methods are detailed hereafter.

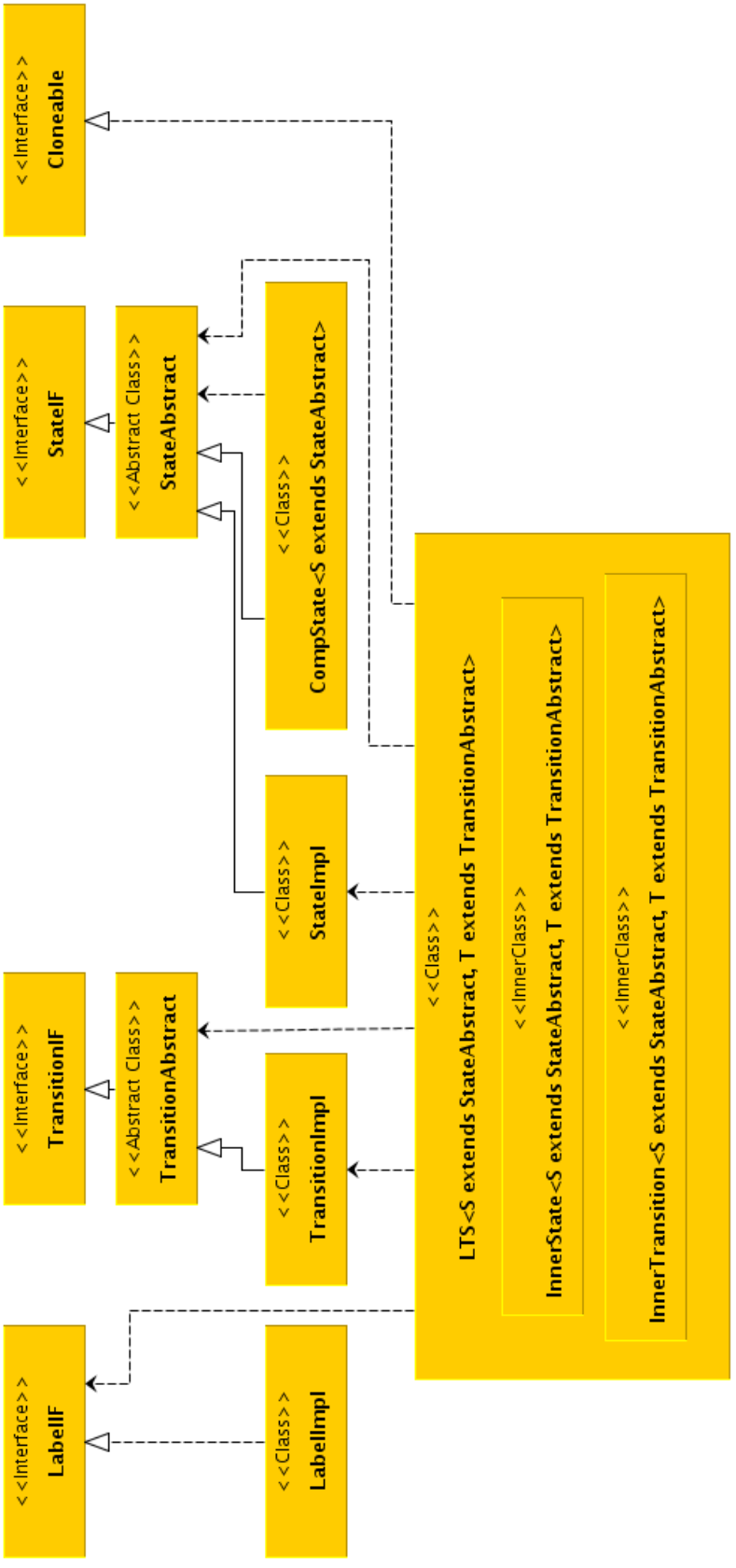

Figure 4.1: UML representation of the `lts` package.

- `LTS()`: This constructor is the simplest one and just initialize the fields of an `LTS` object.
- `LTS(S initialState, Collection<S> states)`: This constructor allows to create an LTS with some predefined states and an initial state which should also be in `states`.
- `LTS(S initialState, Collection<S> states, List<T> transitions, List<S> from, List<S> to)`: Create a new instance given an initial state, a collection of states which has to contain the initial state, and a list of transitions. Two other lists of states are provided. The ith element of `from` is the source of the ith transition and the ith element of `to` is the respective destination of this respective transition.
- `LTS<S, T> clone()`: Duplicate the LTS.
- `void saveAsDotFile (String filename)`: Generate a dot file for the LTS.
- `void addState (S newState)`: Add a new state to the LTS. `newState` cannot be part of the LTS.
- `void removeState(S delState)`: Remove a state from the LTS. `delState` must be part of the LTS.
- `HashSet<T>inTransitions (S s)`: Get the ingoing transitions to a state
- `HashSet<T>outTransitions (S s)`: Get the outgoing transitions from a state
- `List[] getTransitionsFrom(S state)`: Return two lists List<T>,List<S>. The first is the list of transitions in this and the second one is the list of corresponding destination states.
- `List[] getTransitionsTo(S state)` Return two lists List<T>,List<S>. The first is the list of transitions in this and the second one is the list of corresponding source states.
- `List[] getTransitions()`: Return three lists List<T>,List<S>,List<S>. The first is the list of transitions in this, the second one is the list of corresponding source states and the third one is the list of corresponding destination states.
- `Set<T> getTransitionSet()`: Get the set of transitions in the LTS.
- `void addTransition (T newTransition, S fromState, S toState)`: Add a transition to the LTS.
- `void removeTransition (T delTransition)`: Remove a transition from the LTS. `delTransition` must be part of the LTS.
- `boolean hasTransition (T transition)`: Check whether `transition` is already part of the LTS.
- `boolean hasTransition (S from, S to, LabelIF label)`: Check whether a transition with source `from` and destination `to` and labelled `label` is already part of the LTS.
- `S source (T t)`: Get the source state of a transition.
- `S destination (T t)`: Get the destination state of a transition.

The abstract framework

The abstract framework presented above (and represented on Figure 4.1) is a set of interfaces and abstract classes. The class which represents a state of an LTS should extend the `StateAbstract` class provided in the `lts` package. The same mechanism is required for the implementation of a transition and the corresponding abstract class is `TransitionAbstract`. Those two abstract classes implements respectively two interfaces `StateIF` and `TransitionIF` which define some getters the implementations should have in order to be usable by the rest of the framework: `getMode()` and `getName()` for the states, `getLabel()` and `getID()` for the transitions. The abstract classes abstractly override `equals(Object o)` and `hashcoe()` in order to force their overriding when they are extended. For the labels, the `LabelIF` interface should be implemented and doing so the methods `getType()` and `getName()`.

The goal of this abstraction is to hide the internal structure of the LTS to the outside and also to allow a user to personalize its own version of an element of an LTS. Of course, standard implementations are provided. The `StateImpl` class is the one for the states. It just contains two fields, one for the name of the state and one for the mode, and the corresponding getters. This is the simplest implementation. Another implementation has been made for the representation of a composite state: `CompState<S extends StateAbstract>`. This implementation is similar to the previous one except that there are two more fields in it for the corresponding system and mental states. For the transition, the provided implementation is `TransitionImpl`. It also contains two fields, the label and an id. There is an id in order to be able to distinguish two transitions with the same label. Those two implementations also override the `equals` method. Finally, the `LabelImpl` class just has a field for its name and another for its type. The type is defined as an `enum` structure and is there for the distinction between commands and observations inside an LTS.

Inside an LTS instance, the transitions and states are encapsulated in inner types. Figure 4.2 illustrates the relations between internal and external types. The inner types take care of the structure of the LTS. For the `InnerState` class, an instance remembers the in and out transitions. For the `InnerTransition` class, an instance remembers its source and destination states. The fields in those internal classes are of the external types. As mentioned above, each of those inner elements also encapsulates an instance of the respective external class. The correspondence between an external element and its internal encapsulation is kept in the HashMaps presented hereafter, respectively `statesMap` for the states and `transitionsMap` for the transitions. This has indeed to be done because the user only knows about external elements.

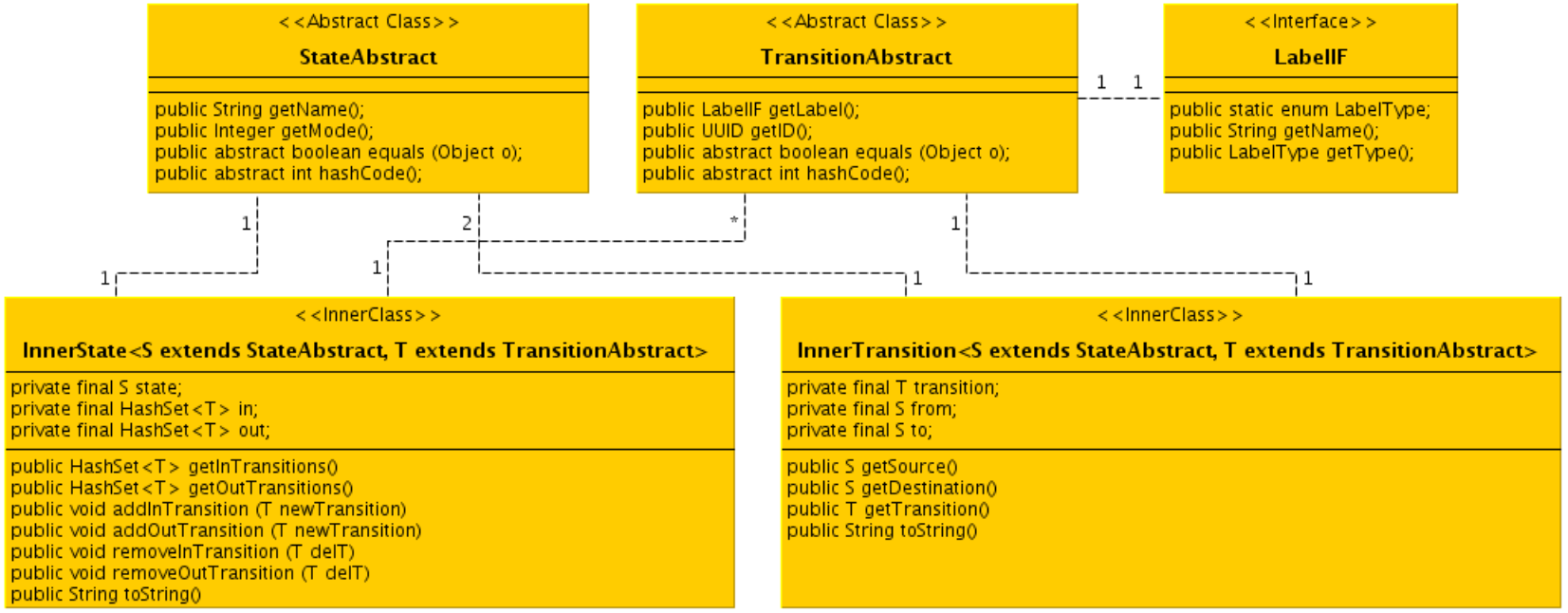

Figure 4.2: Relations between the internal and external representations of the states and transitions.

The `LTS` fields

Here are detailed the data structures in the LTS type. As explained above, those are the internal representations of the different parts of the quadruplet representing a LTS:

- `private HashMap<S, InnerState<S, T» statesMap` corresponds to the $\mathcal{S}$ set. It establishes the binding between the external representation of a state and its internal instance. It is also the collection of all the states in the LTS.
- `private HashSet<LabelIF> labelsSet` corresponds to the alphabet of the LTS ($\mathcal{L}$).
- The initial state s_0 is just kept in `private S initialState`.
- Finally, the transitions collection is like that of the states since the same mechanism is used here for the wrapping in an internal representation of a transition. The corresponding Java one is `HashMap<T, InnerTransition<S, T» transitionsMap`.

The `Cloneable` implementation

Since in the context of the mutants generation, an LTS instance has to be reproduced many times and then modified, it had to be a *cloneable* object. In order to do so, all the types used in it (states, transitions, labels, etc.) had to be *immutable*. An immutable object in Java means that from its creation to its destruction, the state of the object remains the same. Such an object cannot be modified and this gives it the property, among others, to be cloneable as-is. When a collection is cloned, only the reference to the objects within is copied instead of the all content of each object. This means that if an element of the collection resulting of this clonage is modified, the cloned one is also modified. But with immutable objects, since they are not editable, the problem does not subsist. The use of immutable objects allows to clone the collections and considers the clone as a perfect copy of the cloned collection.

This means that if anyone wants to extend the abstract framework explained above, he has to make sure that his own implementation is still immutable. In order to do so here are some guidelines, adapted from [Blo08]. The main rule to follow is: **Don't provide any method that modify the object**. Do not provide any setter class on any field. The following rules are a way to achieve this property which is at the base of the immutable concept:

1. **Make the class final**. This will prevent the class to be extended, this inheritance mechanism allowing to add mutable fields or to overwrite some methods.
2. **Make all fields final and private**. This `final` keyword is the Java way to say that once they are defined, they may no longer be modified. The `private` keyword limits the visibility of the fields. This prevents any user to modify them.
3. **Ensure exclusive access to any mutable component**. It means that if the object has to have a mutable instance as a field, it should be the only one to have access to the reference of this mutable instance. No exports of such a reference are allowed. And if the mutable instance is an external one, then a *defensive copy* should be made. This means that a new instance has to be created with the same value as the original one. And only this copy should be used inside the object.

Saving formats

There are different formats available to save an LTS. The LTS type has the method `saveAsDot-File (String filename)` which is able to save the LTS as a `DOT` file. This is a common language used for graph representation. There are a lot of tools available for the manipulation of `DOT` files. For more information, see [GN00]. The loading from a ".dot" file is not handled since this is a format mainly used for visualization.

As illustration of a ".dot" file, here is the one corresponding to Figure 3.2a.

```
digraph LTS {
        "S1" [shape=box];
        "S1" -> "S2" [label="a"];
        "S2" -> "S3" [label="b"];
}
```

Listing 4.1: ".dot" file corresponding to Figure 3.2a

The `LTSLoader` class is another class from the `lts` package. It provides the loading and the saving of an LTS from its description in a `.lts` file, and the saving of an LTS in a similar file. Of course, such file must have a specific structure. This structure is presented through an example in Listing 4.2. Again, this is the file corresponding to Figure 3.2a.

```
; States
S1      0
S3      0
S2      0
; Transitions
S1      S2      a       0
S2      S3      b       0
```

Listing 4.2: ".lts" file corresponding to Figure 3.2a

In this format, the states are enumerated with their corresponding mode, here 0 for the example. Then each transition is defined with the structure : `[FromStateName] [ToStateName] [Label] [LabelType]`.

4.2 The `mut` package

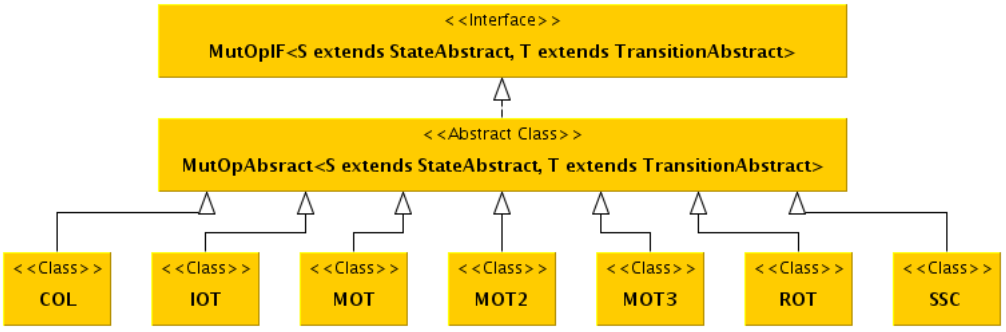

Figure 4.3: UML representation of the `mut` package.

The `mut` package gathers mutation operators. Each operator is a class which extends the `MutOpAbstract` abstract. This abstract class is the common basis between all the mutation operators, it contains some getters and also implements the `MutOpIF` interface.

This architecture of the `mut` package is represented in Figure 4.3. When creating an instance of one of those mutation operators, an LTS is given as input. Then the method `mutate()` implementing the interface's corresponding method provides a new different random mutant each time it is called.

```
1       public LTS<StateImpl, TransitionImpl> mutate();
```

Each mutation operator is base on one or more components of a LTS. The corresponding collection of the elements involved in the mutation is called *components*. If the mutation has to be done on the transitions of an LTS, then *components* is a collection of those transitions either directly retrieved from the LTS itself or constructed from it. Each time the method is called, a element of the collection is chosen at random and the corresponding mutant is returned. Of course, if the mutant is already generated, another one is returned.

Not all mutation operators are only based on one component of an LTS but sometimes on two, or three (IOT for example, insertion of transition). This is why an accumulator mechanism is used in each of the mutation operator. In fact, the call to the `mutate()` method seeds the process with random indices in the range of the different collections of component implied in the mutation. Then, an overload of `mutate` with those indices as parameters is called. This method checks whether the given values correspond to a set of elements on which the mutation has already be done or not. If it is, then the accumulators are incremented and the principle of tail recursion is used by this method to call itself. Otherwise, the corresponding mutant is generated and returned. In order to illustrate this, Algorithm 1 is provided hereafter. It represents the algorithm for only one component involved in the mutation. But then an example will be given with the IOT `mutate` method, and in this later mutation operator, three components are involved.

A cosmetic detail has to be highlighted here. Since states have names and since some mutation operators are brought to modify or to suppress states, it is interesting to give meaningful names at those states. Doing this allows one to visually see the mutation and to have a better understanding of it through this visualization. An example of this can be seen in Figure 5.3 in Section 5.1.2.

The IOT example

As an illustration of the proposed algorithm, the `mutate()` method of the IOT mutation operator is now detailed. As a recall of Section 3.2, this mutation operator inserts a new transition between two existing states of the LTS. It means that for each state considered as a starting one and for each state considered as an ending one, one transition per label is insertable, except for the existing ones.

First, in Listing 4.3 are the declarations of the two collections corresponding to *components* in Algorithm 1, and the one corresponding to *done*. The `done` collection is a map between a pair of states and an array of boolean. This is a two levels table in which there is an entry for each pair of states in the LTS and for each pair there is an array of the length of `labels`. The `Pair` type is just an encapsulation of two states. So for each pair of states, the i^{th} element of the boolean array is set to true when the corresponding mutant has been generated for the i^{th} label in labels.

```
1       private LabelIF[] labels;
2       private StateImpl[] states;
3
4       private HashMap<Pair, boolean[]> done;
```

Algorithm 1: Skeleton of the mutate algorithm

Input: $maxMutateCalls$ a positive integer which represents the maximum number of calls to mutate before having generated all the mutants
Input: acc a positive integer which represents the current number of calls to mutate
Input: $done$ a set of the already done indices
Input: $components$ the collection of the components involved in the mutation
Result: a new random mutant not yet generated by the operator or null if all mutants for this mutation operator have already been calculated.

```
1  mutate():
2  i ← randomValue() ;
3  return mutate(i) ;
4  mutate(int i):
5  acc++ ;
6  if acc ≤ maxMutateCalls then
7  |  if i ∈ done then
8  |  |  return mutate(i + 1);
9  |  else
10 |  |  done ← done ∪ {i} ;
11 |  |  acc ← 0;
12 |  |  return genMutant(components(i));
13 |  end
14 else
15 |  return null;
16 end
```

Listing 4.3: The components collections

Those collections are initialized in the constructor of the mutation operator. The corresponding code is given in Listing 4.4. As we can see IOT is fed with an LTS called `toMutate` this initialization mechanism is the same for all the mutation operators. The two components collections are also retrieved from this LTS. Then, a few variables are initialized. Among them, `maxMutateCall` is initialized as the square of the number of states times the number of labels. This is just the number of possibilities to enter the `mutate` method. The `maxMutants` variable is also available in each implementation of a mutation operator and is the actual number of mutants that will be generated by this operator. The difference between those two numbers is the number of existing transitions in the case of IOT. This is because the mutation operator does not add a duplicate of an existing transition and so, does not output those mutants.

```
1    public IOT(LTS<StateImpl, TransitionImpl> toMutate){
2              this.toMutate = toMutate;
3              name = "IOT";
4
5              labels = toMutate.getLabels().toArray(new LabelIF[0]);
6              states = toMutate.getStates().toArray(new StateImpl[0]);
7
8              acc = 0;
9              maxMutateCalls = states.length * states.length * labels.length;
10             maxMutants = maxMutateCalls - toMutate.getTransitions()[0].size();
11             done = new HashMap<Pair, boolean[]>(maxMutateCalls);
12       }
```

Listing 4.4: The IOT constructor

So the creation of one mutant in IOT involved three components. This is why the `mutate()` method produces three random numbers in the range of the components corresponding collections. The code of the IOT one is given in Listing 4.5.

```
1    public LTS<StateImpl, TransitionImpl> mutate() {
2              Pair i = new Pair((int) Math.floor(Math.random() * states.length), (int) Math.floor(
                   Math.random() * states.length));
3              int j = (int) Math.floor(Math.random() * labels.length);
4              return mutate(i,j,1,1);
5        }
```

Listing 4.5: The IOT `mutate()` method.

Finally, the core of the mutation is provided in Listing 4.6. The signature of the method includes two more accumulators. It is because three components of the LTS are involved in each mutation and because, if for a triplet of indices, the corresponding mutant has already being produced, then the method have to generate the following one. If there was no more accumulator, the other way to proceed would have been to try with another random triplet, but this is inefficient, since when most of the mutants have been generated, the chance to have an already handled triplet raises up and so the pure randomness could loops unnecessarily for a while. Here, for each call, the method outputs a new mutant in a bounded time $\mathcal{O}$(`maxMutateCalls`). The idea is not to have a perfectly random way of producing the mutants, but just to produce them in a well distributed way.

In the method, Algorithm 1 is well followed. Firstly, the method checks whether the main accumulator is still under `maxMutateCalls`. If it is the case, then the method checks whether the mutant has already been generated or not. If not and if the triplet of indices does not correspond to an existing transition the mutant is produced. Otherwise, if the

transition exists or if the mutant has already been returned, a new call is made with the accumulators correctly incremented.

```
1       public LTS<StateImpl, TransitionImpl> mutate(Pair i, int j, int labelAcc,int stateAcc) {
2               acc++;
3
4               boolean[] labelsDone = done.get(i);
5
6               if (acc <= maxMutateCalls){
7                       if (labelsDone==null){
8                               labelsDone = new boolean[labels.length];
9                               labelsDone[j] = true;
10                              done.put(i, labelsDone);
11                      }
12                      else if (labelsDone[j])
13                              if (labelAcc < labelsDone.length)
14                                      return this.mutate(i,(j+1)%labelsDone.length,labelAcc+1,stateAcc
                                            );
15                              else if (stateAcc < states.length){
16                                      Pair iprime = new Pair(i.getA(),(i.getB()+1)%states.length);
17                                      return this.mutate(iprime,j,1,stateAcc+1);
18                              }
19                              else{
20                                      Pair iprime = new Pair((i.getA()+1)%states.length,i.getB());
21                                      return this.mutate(iprime,j,1,1);
22                              }
23                      else
24                              labelsDone[j] = true;
25
26                      LTS<StateImpl, TransitionImpl> mutant = toMutate.clone();
27                      TransitionImpl newTransition = new TransitionImpl(labels[j]);
28                      if (!mutant.hasTransition(states[i.getA()], states[i.getB()], labels[j])){
29                              mutant.addTransition(newTransition, states[i.getA()], states[i.getB()])
                                    ;
30                              acc = 0;
31                              return mutant;
32                      }
33                      else
34                              if (labelAcc < labelsDone.length)
35                                      return this.mutate(i,(j+1)%labelsDone.length,labelAcc+1,stateAcc
                                            );
36                              else if (stateAcc < states.length){
37                                      Pair iprime = new Pair(i.getA(),(i.getB()+1)%states.length);
38                                      return this.mutate(iprime,j,1,stateAcc+1);
39                              }
40                              else{
41                                      Pair iprime = new Pair((i.getA()+1)%states.length,i.getB());
42                                      return this.mutate(iprime,j,1,1);
43                              }
44              }
45              return null;
46      }
```

Listing 4.6: The overload of the IOT `mutate()` method.

4.3 The `checkProp` package

The most important class in this package is the `Checker` class. This class contains the different property checks made on the mutants.

The implementation of the full-control property is straightforward from the definition from Section 3.3.1. The idea is to combine the initial state of the system model and of the mental model in a composite state. This is done in order to make the composition of the two models. Then, the composition is build by following synchronized paths, i.e. transitions available in both corresponding states at the same time. Then we check that in

each of the composite states, the user is able to see at least all the observations going out of the corresponding system model state and do all the commands from the corresponding system model state and just those ones.

The implementation is just a traversal of the composition of the two models which stops as soon as there is a missing observation, a missing command or an extra command. Since it is on this algorithm that all the other are based, the skeleton of it is given in Algorithm 2.

Algorithm 2: Skeleton of the full-control check algorithm

Input: *system* an LTS which represents the system model
Input: *mental* an LTS which represents the mental model

```
1  toExplore = {newCompState(system.initialState, mental.initialState)};
2  while toExplore ≠ {} do
3  |  cur ← toExplore.next();
4  |  toExplore ← toExplore \ {cur};
5  |  if cur ∉ explored then
6  |  |  explored ← explored ∪ {cur};
7  |  end
8  |  if !cond then
9  |  |  doStuff();
10 |  end
11 |  foreach
   |  cmpState ∈ genReachableCmpState(cur.systemstate, cur.mentalstate) do
12 |  |  if cmpState ∉ explored ∧ cmpState ∉ toExplore then
13 |  |  |  toExplore ← toExplore ∪ {cmpState};
14 |  |  end
15 |  end
16 end
```

This algorithm is voluntarily provided with unfilled parts. That is because those parts changed in the alternatives detailed here. For the full-control in itself, `cond` has to be replaced by the condition explained in Section 3.3.1. If this condition is not filled, then it means that *mental* does not have full-control on *system* and `doStuff()` should just return `false`.

Since a mental model having full-control on a system is very sensitive to mutations because of the strength of the property, the mutants do not have many chance of success on this check. This is why some statistics are computed to see to what extent the full-control is lost. The main idea of this approach is to be able to see the coverage of the full-control remaining after a mutation. This is done in two ways. The first one is just to record each of the errors in the traversal of the composition but without stopping it. Doing so, we are able to see which are the missing observations and the missing/extra commands in each composite state. For this algorithm, `cond` is the same has above but if the condition is not filled, then `doStuff()` does not stop the execution but makes a record by filling an `StatTableEntry` with either an extra command, a missing command or a missing observation.

The `StatTableEntry` class is also in this package. This is just a type which allows the algorithm to make some statistics about the errors detected in the full-control checks. It also records the difference between the number of states in the original composition and

in the composition being currently build.

The second way is to use the notion of mode and to stop the traversal on the states where there is mode confusion, i.e. when in a composite state the mode of the corresponding mental state is no longer the same as the one of the system state. This approach allows us to see the border of the full-control. It lets us know to what extent this mutated composition is a sub-graph of the original composition. Nevertheless, this technique has a main drawback which is that it could consider a full-control pair of models being not full-control if the full-control composition contains some mode-confusing composite states. Such states does indeed not lead necessary the system to a loss of full-control. But this will be detected by the other checks and the advantage of this variant is that we can clearly see where there is mode confusion and where the "full-control" zone stops. In this algorithm, `cond` and `doStuff()` are the same as above, but the condition on line 12 of Algorithm 2 is restricted by adding $\wedge$ *cmpState.systemState.mode* == *cmpState.mentalState.mode*. This condition will prevent the algorithm to explore further the graph when a composite state has mode confusion.

The last variant is the alternative full-control. The implementation of this check is again straightforward from the definition given in Section 3.3.2. The only variance with the full-control implementation is that this time, the system state should just contain at least the commands available in the corresponding mental state but on the contrary, a missing command in the mental state is no more a fault with respect to this property. This time, `cond` becomes the condition detailed in Section 3.3.2 and `doStuff()` just return `false` if this condition is not filled.

The reachability property is checked during a traversal. This is just a pattern on the name of a state and this is done by checking if the name of *cur* in the algorithm matches or not the pattern.

4.4 Usage

The usage of `mentalRate` is indeed very simple. The user just has to enter the two ".lts" files containing the description of both the mental and the system model. He may also limit the number of generated mutants for each mutation operator.

When the program has been run by:

```
$ java -jar mentalRate.jar therac-mental.lts therac-system.lts 600
```

An output like the following one is generated for each mutation operator so this is only a snippet of the complete output produced and it is provided as an example. It is also recommended to redirect the standard output to a file.

```
1  therac-mental used as the mental model for the therac-system system model.
2
3  600COL mutants
4  #Extra states (mean in %): 1.27
5
6  #Missing Commands (mean in %): 4.66
7
8  #Extra Commands (mean in %): 4.37
9
10 #Missing Observations (mean in %): 7.08
11
12
13 fullControl: 0
14 statFullControl: 0
```

```
15 mutants StatModeFC: 0
16 alternativeFullControl: 56
17
18 Kills: 54
19
20 [...]
```

The first line appears only once and just recalls the name of the models used. Then the number of generated mutants for the COL mutation operator is given (in this case 600 since the number of possibly generated mutants with those models exceeds this maximum). Then some proportions are provided on the mean percentage of different errors among the mutants. Then, the number of mutant having full-control on the system is given. This number should be equal to the two following ones which are just reports on statistics. At line 16 is given the number of mutant having the alternative full-control on the system. Finally, the number of mutants killing unwillingly the patient is given.

CHAPTER

FIVE

EXPERIMENTS

This chapter is the core of this master thesis. It explains the experiments that were made and the models on which they were made in order to make those experiments reproducible by whoever is interested in the subject and wants to reproduce them. So first, the experiments are explained on a simpler model which is the one of a vehicle transmission system, the gearbox model. Then, the following section looks at the results given by the case study, the Therac-25. An analysis is made on those results in the last section.

5.1 Tests on the gearbox model

This section presents the model on which was made all the testing and validation part of this project. It also provides the different results obtained with this model in order to see what information is produced by the `mentalRate` tool and how to process it.

5.1.1 The model

The gearbox model is a model of the transmission system of a vehicle. As mentioned before, this example is taken from [HD07]. Figure 5.1 is a representation of this model.

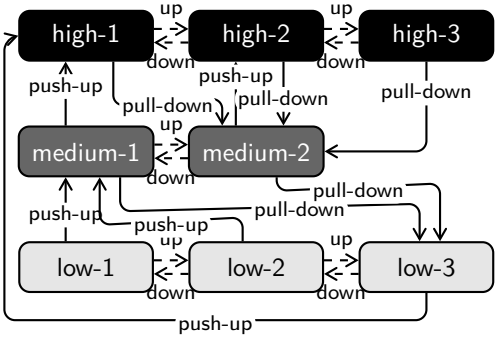

Figure 5.1: The vehicle transmission system example (from [CP09]).

This system contains eight states among which are three distinct operating modes: `low` (in light grey), `medium` (in dark grey) and `high` (in black). The alphabet of this model is composed of two commands: {`pull-down`, `push-up`} and two observations: {`down`, `up`}.

This model is small, but not as simple as it appears to. The corresponding minimal mental model generated by the minimization-based approach presented in Setcion 3.4.1, is not a simple three states LTS with just a “low”, a “medium” and a “high” state because when the user is in the `low` mode, a `push-up` command may lead him either in `medium` either in `high` mode, depending on internal transitions observable by the driver. So, this distinction has to be present in the simplest mental model and this is what makes this model interesting and not so trivial. This simplest mental model is the one presented on Figure 5.2 in which the corresponding modes have been highlighted with the same background colors.

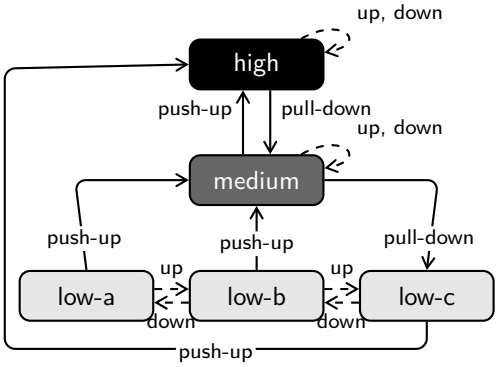

Figure 5.2: The minimal full-control mental model for the vehicle transmission system (from [CP09]).

In order to have some point of comparison, the system can also be considered as a mental model in itself. Of course, a system model may always be considered as a mental model but as discussed earlier, the goal of a mental model is to be an abstraction of the system in order to be simpler.

5.1.2 Preliminary results

Here are provided and explained some results obtained on the gearbox model. The main goal is to show which kind of results are produced and what can be learned from them. This first glance is a way to get familiar with this kind of results before analyzing the data obtained from the Therac-25 model.

In the table 5.1 are provided the number of mutants produced for each mutation operator for each of the two configurations: the minimal mental of Figure 5.2 evaluated with the system and using the system model as its own mental model. In this table are also provided the amount of mutants that passed the full-control property (FC in the table) and the alternative full-control property (Alt-FC in the table).

Quick analysis

In this point are exposed the reasons which make that a specific mutant passes the FC or the Alt-FC test. Those reasons are dependent on the structure of the graph of the considered mutated model. And since those specifics structures are often found in other models, those reasons are generic and the related explanations apply also for the case study and any other application.

In the case of the COL (change of label) operator, the mutants succeed the alternative full-control when a label representing a command is replaced with a label representing an

Mental used	Minimal mental model			System model		
# States	5			8		
# Transitions	14			20		
	# mutants	# FC	# Alt-FC	# mutants	# FC	# Alt-FC
COL	42	0	2 (4.8%)	60	0	8 (13.3%)
IOT	86	10 (11.6%)	10 (11.6%)	236	48 (20.3%)	48 (20.3%)
MOT	14	0	6 (42.9%)	20	0	10 (50%)
MOT2	14	0	0	20	0	0
MOT3	14	0	0	20	0	1 (5%)
ROT	14	4 (28.6%)	5 (35.7%)	20	0	0
SSC	4	0	0	7	0	0

Table 5.1: Amount of mutants generated for each type of mutation operator.

observation not yet going out of the corresponding mental state. This produces indeed a missing command in the mental model, making it fail the full-control test. However, since in the alternative version of this property the mental is not required to have all the commands, the test passes for this mutant. If the label was changed into an already outgoing label, then it would have induced non-determinism in the composition (which is not prohibited) and this increase the chances of loosing full-control. If the non-determinism lead to a composite state where the mental and the system are not full-control equivalent, then the full-control does not hold any more.

For IOT (insertion of transition), we can observe for the first time full-control mutants. We can see that the number of mutants succeeding Alt-FC is the same that the number succeeding FC and this in the two experiments. This is not accident, the mutants succeeding FC are the same as those succeeding Alt-FC. This is due to the definition of Alt-FC which is, as mentioned in Section 3.3.2, a weaker version of FC.

For the MOT (missing of transition) operator, the mutants succeeding Alt-FC are just corresponding to the mutated model with one command having been removed. By definition of Alt-FC, again, the user is not forced to do any command.

The MOT3 mutant succeeding the Alt-FC check is not a generic case. The corresponding LTS is drawn on Figure 5.3. The mutation in this case has been the removing of the `pullDown` transition between the `medium1` and the `low3` state. The result is a merged state between those two mental states. This mutant does not passes the FC check since the `pullDown` command is not anymore available from the corresponding mutated mental state. But since in the Alt-FC, the mental is not forced to have all commands, the check passes. This is also due to the fact that the same observations are available in both modes, otherwise there would have been some missing observations.

The four ROT (reverse of transition) mutants succeeding the FC check are a kind of trivial ones. They correspond indeed to the reversal of the self-loops representing the observations on the medium and high mental model. These mutants are produced because self-loops are not detected by the mutation operator. This is an example of equivalent mutants.

However, the fifth one, succeeding only the Alt-FC check is more interesting. It succeeds because one part of the LTS is isolated following the reversal of the transition. Since the only way out was a command and that it is not a fault if this command misses, the mutant succeeds the test. This mutant is drawn on Figure 5.4. In fact, it is even more complicated since the reversed command must have been one with a label already going

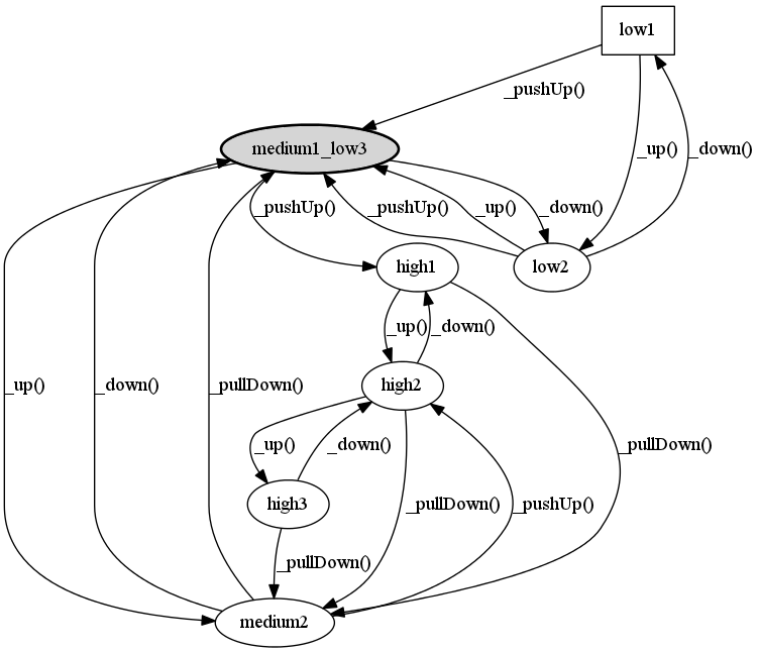

Figure 5.3: The MOT3 mutant succeeding Alt-FC.

out of the medium state, otherwise there would have been an extra command which is a

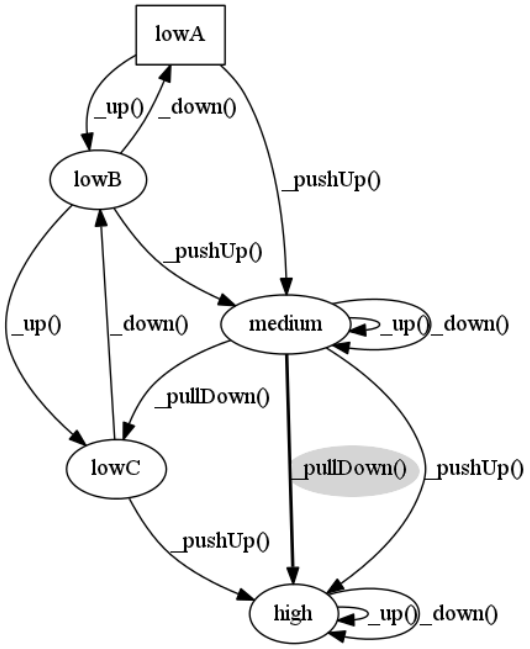

Figure 5.4: The ROT mutant succeeding only Alt-FC.

fault to the full-control property.

Some generated models

As an illustration of the kind of models produced by the tool, here are provided some LTSs which represent the outputs of the different checks made for the full-control property as explained in Section 4.3. Figure 5.5 represents a COL mutant, where the label of the `down` transition from `lowB` to `lowA` has been changed in `pushUp`. This mutated mental will of course fail the check on the full-control and alternative full-control property because of the non-determinism generated by two outgoing transitions with the same label from an unique state and also because of the missing `down` observation. The composition stopped at the first mode confusion is shown at Figure 5.6 while the full composition is provided in Figure 5.7. The composite states with mode confusion have a diamond shape. What is interesting to observe is the comparison between those two compositions which allows to see where exactly the mental fails and how the user is able to misunderstand and misuse the system. An other interesting observation to make is that in Figure 5.7, there exists a path, through mode-confusing composite states, which comes back to the full-control zone (the oval shaped states). This means that, even with a wrong mental model, the user is able to do a valid sequence of actions without problem.

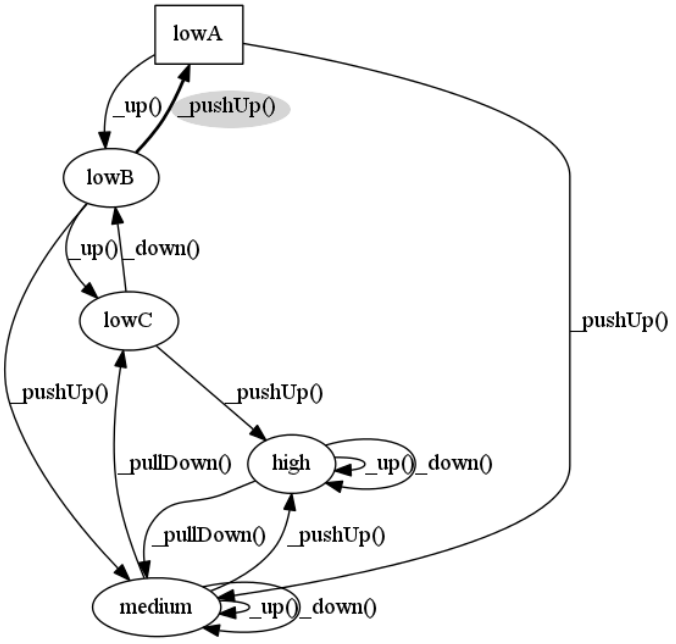

Figure 5.5: A COL mutated mental.

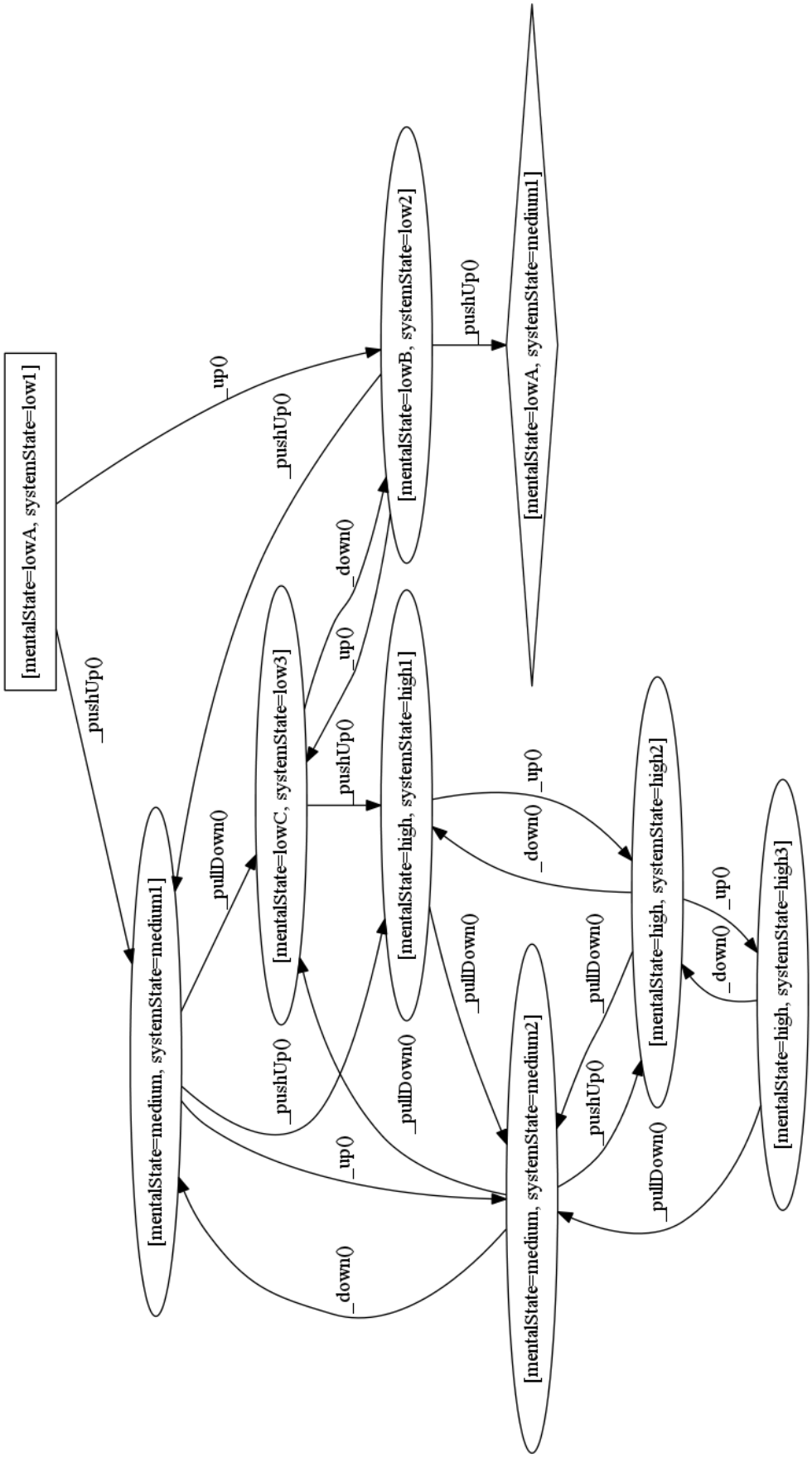

Figure 5.6: The composition of a COL mutated mental with the system model stopped when there is mode confusion.

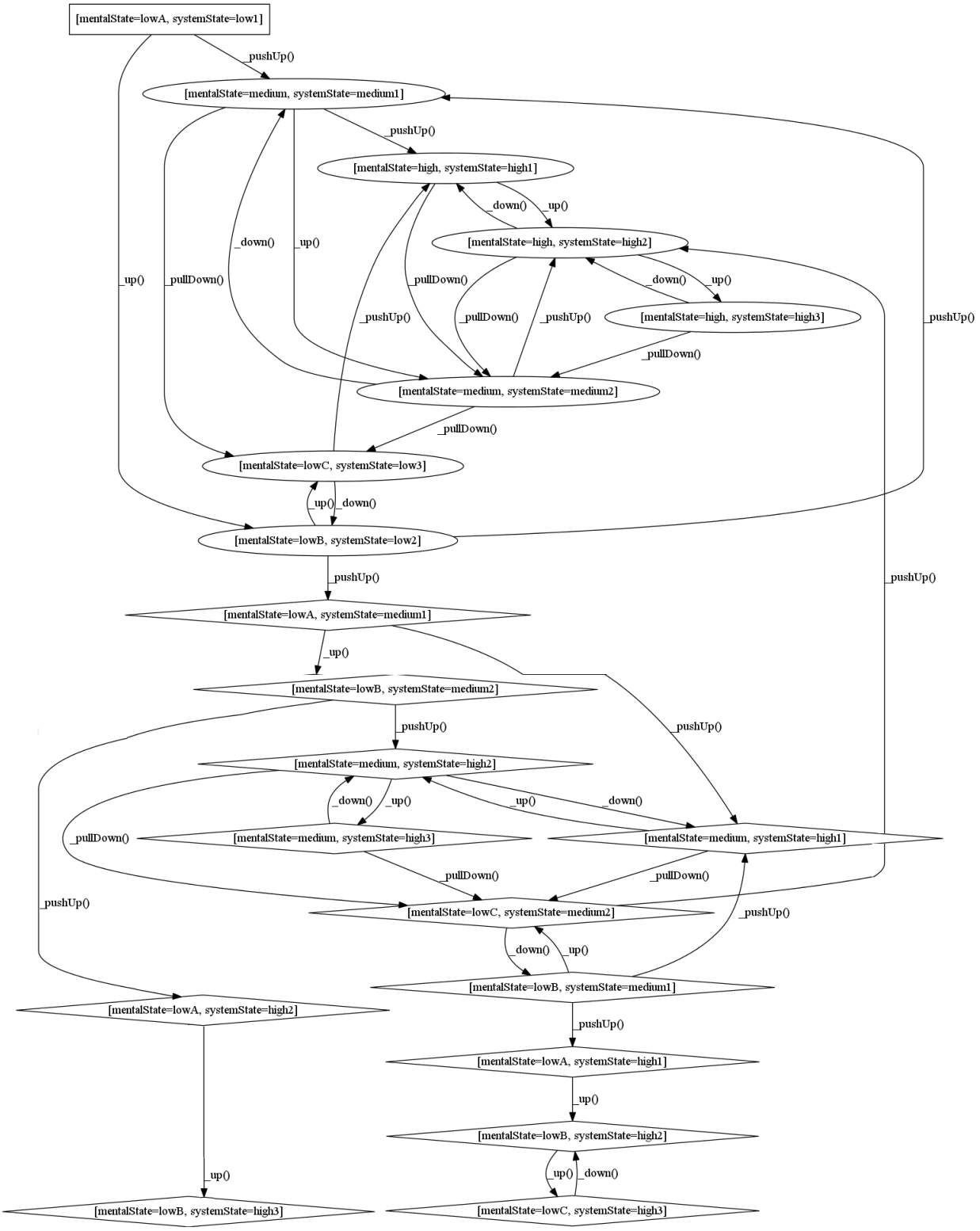

Figure 5.7: The composition of a COL mutated mental with the system model.

5.2 Case study: the Therac-25

This section is about the presentation of the main example of this master thesis. The experiments are made with the Therac-25 model which is used as a case study. The results are quite interesting since they show that the idea at the origin of this master thesis was a good one and brings some real opportunities. The aim of this case study is to see whether the tools and techniques developed in the scope of this master thesis are well founded and give results on a "real-life" model.

First, the model and the mental models used are presented. Then the results are analyzed with two different approaches: an analysis of the mutants generated and then an analysis of the results of the `checkProp` tool.

5.2.1 Presentation of Therac-25

The Therac-25 was a medical device which caused the death of patients due to flaws in the design of the interface. This machine has been widely studied in the literature, see for example [LT93] or [BBS08].

The Therac-25 is a radiation therapy machine able to operate in two modes. In the first one, the machine administers electron beams, which deliver high-energy electrons (between 5 and 25 MeV). The second is the X-ray mode which is intended to work deeper in the tissues and use high-energy collisions (25 MeV) between electrons in order to produce the X-rays. The important difference between those modes, which is at the origin of the reputation of the device, is that in X-ray mode, the machine puts a beam flattener in the path of the X-rays in order to spread the treatment over a uniform area. This spreader is not used in the electron beam mode and an administration of X-rays without this spreader may harm or even kill the patient. This mechanism is illustrated in Figure 5.8.

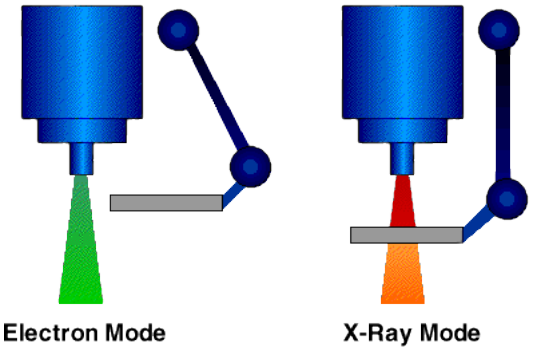

Figure 5.8: The two operational modes of the Therac-25 (from [Gal11]).

A formal model of the system has been proposed in [BBS08] on which is based the one used in this master thesis. It is provided as a state chart at Figure 5.9. On this picture, the interface proposed to the administrator is shown on the left while the actual device model is on the right. The alphabet of this system is simple. Each of the commands were available on the keyboard of a computer outside of the treatment room and this is why to each command corresponds a keystroke. The "x" keystroke is the `selectX` command and allows the user to switch to the X-ray mode. The corresponding key for the electron mode is "e" for `selectE`. The `up` command allows the administrator to cancel its last action in the planning of the treatment, the corresponding key is ↑. The last command is the `fire` one and the corresponding key is ↵. This last command allows to administer the planned

treatment. To complete the alphabet, there are also two observations, the `reset` one which happens after a firing in the system and could correspond to the return key after pushing "b" and an observation `8seconds` which corresponds to an automatic timeout after which the switch between the two modes is actually made. This immediately highlights the flaw, since this observation is not shown in the interface.

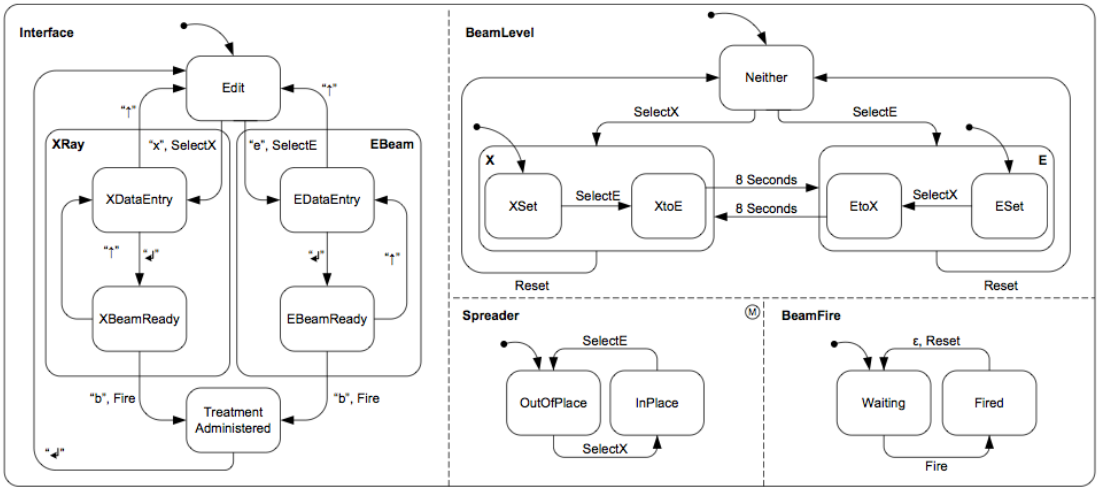

Figure 5.9: The formal model of the Therac-25 from [BBS08].

From this model, the problem is that the administrator can not observe the change of mode when eight seconds pass. This results in a mode confusion. The administrator thinks he is still in a mode while indeed he is in the other one. This is due to an automatic reset of the system after eight seconds. So, if the administrator does the following sequence in less than this timeout: `selectE`, `up`, `selectX`, `fire`, the X-ray is sent without the spreader. This sequence corresponds to the administrator choosing the wrong mode, pushing the ↑ key to cancel and pressing "x" to go in the wished mode. The system is then either in X-ray or in electron beam mode depending on the timeout.

5.2.2 Different mental models

Based on the model presented above, the system used is an enriched version of the Therac-25 model. In fact, mode observations had been added to the system in order to model the fact that we want the user to know at each moment in which mode the system operate. To do so, self-loops have been added in the mental states. But since, in the full-control property, we want to enforce the modes being known, those self-loops are treated as commands. The intuition is that those transitions have to be there otherwise it would mean that the user does not know the mode.

For this system model, two mental models have been created. Both were generated from the system model using the learning-based variant described in Section 3.4.2.

In the following, we call "first" the mental model corresponding to the upper bound of the learned 3DFA and its minimization is called the "second" mental. The first is made of 64 states and 224 transitions while the second one is made of 24 states and 102 transitions.

The goal of having two different mental models for the same system model is to have different levels of abstraction with respect to the system. Again, since the aim of this master thesis is to be able to rate the robustness of a mental model, it is very interesting

to be able to compare the results for different mental models and verify whether a more redundant model is indeed well evaluated.

5.2.3 Results

In Table 5.2 and Table 5.3 are provided the results of the analysis on the different properties for each mutation operator, respectively for the first and the second mental model. As a recall, the analyzed properties for the case study are the full-control, the alternative version of the full-control and a reachability property. This last property aims to see whether the mutants allow to kill a patient unintentionally. This is done by traversing the composition of a mutant with the system model and searching for a state where there is mode confusion and where the spreader is out of place, the beam mode is X-ray and the beam has been fired.

As explained before, the number of possible mutants for the COL and IOT mutation operators are big and so the generation is limited to 600 mutants of each type.

Mental used	Mental model 1			
# States	64			
# Transitions	224			
	# mutants	# FC	# Alt-FC	# Kills
COL	600	0	57 (9.5%)	49 (8.2%)
IOT	600	66 (11%)	66 (11%)	64 (10.66%)
MOT	224	0	154 (68.7%)	0
MOT2	224	0	44 (19.6%)	50 (22.3%)
MOT3	224	0	44 (19.6%)	74 (33%)
ROT	224	44 (19.6%)	64 (28.6%)	0
SSC	63	5 (7.9%)	5 (7.9%)	0

Table 5.2: Results for the first mental model for the Therac-25.

Mental used	Mental model 2			
# States	24			
# Transitions	102			
	# mutants	# FC	# Alt-FC	# Kills
COL	600	0	12 (2%)	99 (16.5%)
IOT	600	15 (2.5%)	15 (2.5%)	86 (14.3%)
MOT	102	0	58 (56.9%)	0
MOT2	102	0	16 (15.7%)	36 (35.3%)
MOT3	102	0	16 (15.7%)	47 (46.1%)
ROT	102	24 (23.5%)	30 (29.4%)	24 (23.5%)
SSC	23	1 (4.3%)	1 (4.3%)	0

Table 5.3: Results for the second mental model for the Therac-25.

At first glance, the proportion of mutants passing the checks is greater for most of the mutation operators for the first mental model while the proportion of kills is smaller. This is what should be expected of a more robust mental model.

As mentioned in Section 5.1.2, most of the patterns found in the results of the gearbox models are still visible here and the explanations, which we supposed to be generic, are

indeed also relevant in the Therac-25 case. Some differences are nevertheless observable. Among them there are some new categories of mutants passing the tests.

In the gearbox results, there were no MOT2 mutant succeeding the Alt-FC test. With the Therac-25 results there are some. Moreover in exactly the same proportion as with the MOT3. There is no coincidence here, this is due to the self-loops of the modes. Indeed, the MOT2 and MOT3 mutation operators applied on a self-loop transition do not modify any other part of the graph. So, they just remove the transition in this case and this produces a mutated mental model that is still Alt-FC, since the user is allowed to have missing commands.

Another difference is that now, there are also SSC (starting state changed) mutants passing the tests. The main reason to this is because there are some equivalent initial states in the system from which the user can get back in the official initial system state without any confusion.

Regarding the kills, we can also see that the proportion of mutants killing the patient in an undesired way is greater in the second mental model.

5.2.4 `checkProp` analysis

As described in Section 4.3, some statistics are also computed during the different checks being made. Those statistics represent the mean amount of missing/extra states, missing/extra commands, and missing observations among the mutants. Table 5.4 provides those statistics for the second mental model test.

	# Missing States	# Missing Commands	# Extra Commands	# Missing Obs
COL	-0.9	7.02	6.66	1.48
IOT	-1.45	5.61	6.3	0.25
MOT	0.96	0.98	0	0.98
MOT2	-4.54	16.72	14.68	2.98
MOT3	14.77	18.4	20.53	1.37
ROT	-0.6	8.49	8.06	1.81
SSC	95.42	0.19	0.16	0

Table 5.4: Statistics for the second mental model for the Therac-25. The values are percentage. In the "Missing States" column, a negative value means that there were extra states. The three other columns are mean values among all the mutants and a value for a mutant is the average value for all its states.

Values near from zero are frequent; their meaning is that the mutated models are not far from a full-control. At least, they are interpreted like this in correlation with the other results. It is also interesting to see the impact of MOT2 and MOT3 which induce a lot of mess in the commands. The last interesting number is the huge amount of missing states in the SSC mutants since when an initial state is changed in such a way that the mental can not handle it, then all the rest of the composition is lost. This reflects well the results obtained above for this mutation operator.

5.3 Analysis

This section puts the different concepts in perspective with the assumptions of this master thesis in regard of the results obtained. First, the choice of the mutants will be discussed, then the properties and finally the robustness rate.

5.3.1 About the mutants

The chosen mutation operators model knowledge imperfections of a user in front of a system. The significance of those has already been discussed. But after seeing the results, the relevance of each of them is to be discussed. The criteria for a mutation operator to be relevant in the scope of this master thesis is its generated mutants passing the checks involving a certain robustness of the system. It is also the highlighting of nuanced results.

The COL and IOT mutation operators are both interesting ones, since they bring a lot of interesting results. The interpretations made about the corresponding operation errors still leads to mutants passing the FC and Alt-FC tests and this means that the system is sufficiently robust to manage such imperfections in some cases. Nevertheless, the main drawback of those mutation operators is the potentially huge amount of mutants generated.

The MOT mutation operator seems too trivial to produce interesting results. While the MOT3 one has shown in the different cases some interesting results. It has the advantage of being a good representation of a forgotten transition.

The ROT mutation operator also brought some good results especially when isolating a part of the graph and allowing to pass the Alt-FC test. The interpretation on the robustness of the system made above also holds for this mutation operator

And finally, regarding the SSC mutation operator, the results were interesting when the system was sufficiently flexible and observable to allow a user in a wrong starting state to handle the mutation and get full-control nevertheless.

5.3.2 About the properties

We have often seen that the full-control property is very strong. Indeed, an initial concern when we started this work was all the mutants failing this check because of it. As it turns out, this property is very interesting since passing its check means a lot in terms of interpretation. It means that the system model is robust enough to be able to manage operation errors of the user in some case.

Since this property check is a little bit radical, statistics have also been made. And most of the mutant not passing the full-control test have nevertheless a quite good coverage of the composition on average. A lot of statistics were close to zero and not very relevant but the ones for the MOT2, MOT3 and SSC mutants allows us to see that those mutation operators were the most devastating on the mental model.

The idea of the alternative full-control is also very interesting since it brought more successful results for the mutants and it represents more accurately the “real world” since this property lets the user chose to not make a command.

Regarding the last property studied, which is the reachability for the killing composite states, the observations made are also in the sense of a good evaluation of the robustness of a mental model. This property highlighted the fact that, in this specific case, a wrong composite state was on average less reachable with a more robust system.

5.3.3 About the robustness

The whole point of this master thesis was to elaborate an evaluation of the robustness of a mental model with respect to some properties and mutation operators. The process which has been proposed answers to this problematic, not by giving an actual rate but by providing some results which still need to be analyzed. The results are the succeeding rates of the mutants again meaningful properties and the analysis which needs to be done is see whether the succeeding rates are sufficiently high to evaluate the system as sufficiently robust.

In this master thesis, we compared each time two mental models for the same system in order to have a point of comparison. And this highlighted that the process was indeed well designed. A user of this framework would have to defined his own acceptance threshold to fulfill his robustness requirements based on the outputs of the tool for its mental model.

CONCLUSION

The main goal of this master thesis was to see if it was possible to associate human-machine analysis and mutation-based testing in order to evaluate the robustness of mental model facing manipulation errors. The conclusions are that this is indeed doable and this document provide a background, a framework and discussions about the way to implement this association.

The main results of this thesis are:

- A set of mutation operators which are detailed and studied. Those mutation operators have been chosen among the literature and based on common sense. A further step in this direction could be to study how well they model actual operation errors, to find a way to validate them as good representation of human misuse, to enrich the set with other mutation operators modeling other imperfections. Another direction to improve this set could be to use the amelioration techniques detailed for mutation operators in general in Section 2.1 and applied them to this set in order to refine the selection. The technique appearing to be the most promising seems to be the mutant reduction since some mutation operators generate a huge amount of mutants.

- A methodology proposed to study the robustness of a mental model. Some properties chosen among existing ones and new ones have been used in order to be able to evaluate the quality of the mutants. This methodology has been tested through the gearbox model and the Therac-25 model. The results have been analyzed and prove that there is a real opportunity in this approach. It has indeed been shown that more redundant mental models were better able to resist manipulation errors, as intended at the beginning of this work. To enforce those results, a further application could be to test the difference between a system and a modified version of this system model with added redundancy in order to confirm the analyze made in this work.

- An implemented framework which is able to produce data to be analyzed in a described way in order to evaluate the robustness of a given mental model. This framework is extensible and allows to generate mutants according to a given set of mutation operators detailed in Section 3.2 and to check the properties described in Section 3.3 on the generated mutants. The output of this process is statistics about the passing rate of the mutants against those properties.

- A major learning for me as a student. I indeed had a lot to learn through this year's work. Having a whole project on my own, managing the time, discovering new theoretical fields through articles and a lot of other things related to redaction, the

implementation and conclusion of such a work have been difficult tasks to overcome but not without work and pride.

BIBLIOGRAPHY

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples* 1. *Information and computation*, 75(2):87–106, 1987.

[BBB90] G. Britain, A.A.I. Branch, and G. Britain. *Report on the Accident on Boeing 737-400 G-OBME Near Kegworth, Leicestershire on 8 January 1989*. HMSO, 1990.

[BBS08] ML Bolton, EJ Bass, and RI Siminiceanu. Using formal methods to predict human error and system failures. 2008.

[BDLS80] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233. ACM, 1980.

[BG06] H. Bowman and R. Gomez. *Concurrency theory: calculi and automata for modelling untimed and timed concurrent systems*. Springer-Verlag New York Inc, 2006.

[Blo08] J. Bloch. *Effective Java*. Prentice-Hall PTR, 2008.

[Byb03] M. Bybro. A mutation testing tool for java programs. *Master's thesis, Stockholm University, Stockholm, Sweden*, 2003.

[CFC+09] Y.F. Chen, A. Farzan, E. Clarke, Y.K. Tsay, and B.Y. Wang. Learning minimal separating dfa's for compositional verification. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, 2009.

[CGP] E.M. Clarke, O. Grumberg, and D. Peled. Model Checking. 2000.

[CH03] J.C. Campos and M.D. Harrison. From hci to software engineering and back. *Bridging the Gaps Between Software Engineering and Human-Computer Interaction*, page 49, 2003.

[CH08] J. Campos and M. Harrison. Systematic analysis of control panel interfaces using formal tools. *Interactive Systems. Design, Specification, and Verification*, pages 72–85, 2008.

[CHL04] J.C. Campos, M.D. Harrison, and K. Loer. Verifying user interface behaviour with model checking. In *Proceedings of the 2nd International Workshop on Verification and Validation of Enterprise Information Systems*, pages 87–96. Citeseer, 2004.

[CP09] S. Combéfis and C. Pecheur. A bisimulation-based approach to the analysis of human-computer interaction. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, pages 101–110. ACM, 2009.

[CPGF] Sébastien Combéfis, Charles Pecheur, Dimitra Giannakopoulou, and Michael Feary. Learning system abstractions for human-machine interactions. Not yet published.

[CRB07] P. Curzon, R. Rukšėnas, and A. Blandford. An approach to formal verification of human–computer interaction. *Formal Aspects of Computing*, 19(4):513–550, 2007.

[DCH00] G.J. Doherty, J.C. Campos, and M.D. Harrison. Representational reasoning and verification. *Formal Aspects of Computing*, 12(4):260–277, 2000.

[Deg04] A. Degani. *Taming HAL: Designing interfaces beyond 2001*. Palgrave Macmillan, 2004.

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick Gerald Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[FDMM02] P.F. Fabbri, ME Delamaro, JC Maldonado, and PC Masiero. Mutation analysis testing for finite state machines. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 220–229. IEEE, 2002.

[Gal11] T. Gallagher. Therac-25 computerized radiation therapy. http://www.kellyhs.org/itgs/ethics/reliability/THERAC-25.htm, May 2011.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[Ham77] R.G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, pages 279–290, 1977.

[HD07] M. Heymann and A. Degani. Formal analysis and automatic generation of user interfaces: Approach, methodology, and an algorithm. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 49(2):311, 2007.

[JH10] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions of Software Engineering*, To appear, 2010.

[JP] D. Javaux and P.G. Polson. A method for predicting errors when interacting with finite state machines. *Javaux and De Keyser [5]*.

[KO91] K.N. King and A.J. Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[LDL09] J. Li, G. Dai, and H. Li. Mutation Analysis for Testing Finite State Machines. In *2009 Second International Symposium on Electronic Commerce and Security*, pages 620–624. IEEE, 2009.

[Lip71] R. Lipton. Fault Diagnosis of Computer Programs. *Student Report, Carnegie Mellon University*, 1971.

[LT93] N.G. Leveson and C.S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[OAL07] J. Offutt, P. Ammann, and L. Liu. Mutation testing implements grammar-based testing. In *Mutation Analysis, 2006. Second Workshop on*, page 12. IEEE, 2007.

[OL91] A.J. Offutt and S.D. Lee. How strong is weak mutation? In *Proceedings of the symposium on Testing, analysis, and verification*, pages 200–213. ACM, 1991.

[ORZ93] A.J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th international conference on Software Engineering*, pages 100–107. IEEE Computer Society Press, 1993.

[OU01] A.J. Offutt and R.H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, page 44. Kluwer Academic Publishers, 2001.

[Rus02] J. Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering & System Safety*, 75(2):167–177, 2002.

[TG08] H. Thimbleby and J. Gow. Applying graph theory to interaction design. *Engineering Interactive Systems*, pages 501–519, 2008.

[Wah95] KS Wah. Fault coupling in finite bijective functions. *Software Testing, Verification and Reliability*, 5(1):3–47, 1995.

[Wah00] K.S.H.T. Wah. A theoretical study of fault coupling. *Software testing, verification and reliability*, 10(1):3–45, 2000.

[WM95] W.E. Wong and A.P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

Appendices

APPENDIX

A

API OF THE LTS PACKAGE

For more information about those implementations, read Section 4.1

A.1 Interfaces

A.1.1 LabelIF

Enum

Define the different available types of label
`public static enum LabelType;`

Methods

Get the name of the label
Pre:-
Post: The returned value contains the name of the label
`public String getName();`

Get the type of the label
Pre:-
Post: The returned value contains the type of the label
`public LabelType getType();`

A.1.2 StateIF

Methods

Get the name of the state
Pre:-
Post: The returned value contains the name of the state

```
public String getName();
```

Get the type of the state
Pre:-
Post: The returned value contains the mode's value of the state
```
public Integer getMode();
```

Override
```
public String int toString();
```

A.1.3 `TransitionIF`

Methods

Get the label associated with the transition
Pre:-
Post: The returned value contains the label associated with this transition
```
public LabelIF getLabel();
```

Get the unique identifier of the transition
Pre:-
Post: The returned value contains the unique identifier of the transition
```
public UUID getID();
```

A.2 Abstract classes

A.2.1 `StateAbstract`

implements `StatelIF`

Methods

Override
```
public abstract boolean equals (Object o);
```

Override
```
public abstract int hashCode();
```

A.2.2 `TransitionAbstract`

implements `TransitionIF`

Methods

Override
```
public abstract boolean equals (Object o);
```

Override
```
public abstract int hashCode();
```

A.3 Classes

A.3.1 `LabelImpl`

implements `LabelIF`

Fields

```
private final String name;
private final LabelType type;
```

Constructor

Pre: name, type != null
Post: An instance of this is created, representing a label with specified name and type
```
public LabelImpl (String name, LabelType type);
```

Methods

Overrides
```
public boolean equals (Object o);
```

Overrides
```
public int hashCode();
```

A.3.2 `StateImpl`

extends `StateAbstract`

Fields

```
private final String name;
private final Integer mode;
```

Constructors

Pre: name, mode != null
Post: An instance of this is created, representing a state with specified name and mode
`public StateImpl (String name, Integer mode);`

Pre: name != null
Post: An instance of this is created, representing a state with specified name
`public StateImpl (String name);`

Methods

Override
`public String int toString();`

A.3.3 `CompState<S extends StateAbstract>`

extends `StateAbstract`

Fields

```
private final String name;
private final Integer mode;
private final S systemState;
private final S mentalState;
```

Constructors

Pre: systemState, mentalState != null
Post: An instance of this is created, representing a composite state with specified system state and mental state
`public CompState(S systemState, S mentalState)`

Methods

Get the system state
Pre:-
Post: The returned value contains the system state
`public S getSystemState();`

Get the mental state
Pre:-
Post: The returned value contains the mental state

```
public S getMentalState();
```

Override
```
public String int toString();
```

A.3.4 `TransitionImpl`

extends `TransitionAbstract`

Fields

```
private final LabelIF label;
private final UUID id;
```

Constructors

Pre: label != null
Post: An instance of this is created, representing a transition with specified label
```
public TransitionImpl(LabelIF label)
```

Methods

Override
```
public String int toString();
```

A.3.5 LTS<S extends StateAbstract, T extends TransitionAbstract>

implements Cloneable

Fields

```
private S initialState;
private HashSet<LabelIF> labelsSet;
private HashMap<S, InnerState<S, T» statesMap;
private HashMap<T, InnerTransition<S, T» transitionsMap;
private int nbrCmds = 0, nbrObs = 0;
```

Constructors

Pre: -
Post: An instance of this is created, representing an LTS
```
public LTS();
```

Pre: initialState, states != null, initialState ∈ states
Post: An instance of this is created, representing an LTS with specified states and initial state
`public LTS(S initialState, Collection<S> states);`

Create a new instance given an initial state, a collection of states which has to contain the initial state and a list of transitions. Two other lists of states are provided. The ith element of from is the starting state of the ith transition and the ith element of to is the respective ending state of this respective transition.
Pre: initialState, states, transitions, from, to != null, initialState ∈ states
Pre:
Post: An instance of this is created, representing an LTS with specified states and initial state
`public LTS(S initialState, Collection<S> states, List<T> transitions, List<S> from, List<S> to);`

Methods

Add a new state to the LTS. ***`newState`*** *cannot be part of the LTS.*
Pre: newState != null, newState ∉ LTS
Post: newState has been added to the LTS. If this LTS was empty, newState has been set to be the initial state of this LTS
`public void addState (S newState);`

Remove a state from the LTS. ***`delState`*** *must be part of the LTS.*
Pre: newState != null, newState is a state of this LTS
Post: delState has been removed from the LTS.
`public void removeState(S delState);`

Add a transition to the LTS.
Pre: newTransition, fromState, toState != null
Pre: fromState, toState are states of this LTS
Post: The returned value contains the label associated with this transition
`public void addTransition (T newTransition, S fromState, S toState);`

Remove a transition from the LTS. ***`delTransition`*** *must be part of the LTS.*
Pre: delTransition != null
Post: delTransition has been removed from the LTS.
`public void removeTransition (T delTransition);`

Check whether ***`transition`*** *is already part of the LTS.*
Pre: transition != null
Post: The returned value is true if transition ∈ the LTS and false otherwise.
`public boolean hasTransition (T transition);`

Check whether a transition with source ***`from`*** *and destination* ***`to`*** *and labelled* ***`label`*** *is already part of the LTS.*

Pre: from, to, label != null, from and to are in the LTS
Post: The returned value is true if there is a transition from `from` to `to` labelled `label`.
`public boolean hasTransition (S from, S to, LabelIF label);`

Get the initial state of the LTS.
Pre: -
Post: The returned value contains the initial state of the LTS
`public S getInitialState();`

Get the number of commands in the LTS.
Pre: -
Post: The returned value contains the number of commands in the LTS.
`public int getNbrCmds();`

Get the number of observations in the LTS.
Pre: -
Post: The returned value contains the number of observations in the LTS.
`public int getNbrObs();`

Get the states of the LTS
Pre: -
Post: The returned value contains the states of the LTS
`public Collection<S> getStates();`

Get the labels of the LTS.
Pre: -
Post: The returned value contains the labels of the LTS.
`public Set<LabelIF> getLabels();`

Get the transitions of the LTS.
Pre: -
Post: The returned value contains the transitions of the LTS.
`public Set<T> getTransitionSet();`

Get the transitions of the LTS with the corresponding source and destination states
Pre: -
Post: Return three lists List<T>,List<S>,List<S>. The first is the list of transitions in this, the second one is the list of corresponding source states and the third one is the list of corresponding destination states.
`public List[] getTransitions();`

Get the ingoing transitions to state with the corresponding source states.
Pre: state != null, state is a state of the LTS
Post: Return two lists List<T>,List<S>. The first is the list of transitions in this and the second one is the list of corresponding source states.
`public List[] getTransitionsTo(S state);`

Get the outgoing transitions from state with the corresponding destination states.
Pre: state != null, state is a state of the LTS
Post: Return two lists List<T>,List<S>. The first is the list of transitions in this and the second one is the list of corresponding destination states.
```
public List[] getTransitionsFrom(S state);
```

Get the ingoing transitions from a state
Pre: s != null, s is a state of this LTS
Post: The returned value contains the set of transitions whose destination state is the specified one
```
public HashSet<T>inTransitions (S s);
```

Get the outgoing transitions from a state
Pre: s != null, s is a state of this LTS
Post: The returned value contains the set of transitions whose source state is the specified one
```
public HashSet<T>outTransitions (S s);
```

Get the source of a transition.
Pre: t != null, t is a transition of this LTS
Post: The returned value contains the source state of the specified transition.
```
public S source (T t);
```

Get the destination of a transition.
Pre: t != null, t is a transition of this LTS
Post: The returned value contains the destination state of the specified transition.
```
public S destination (T t);
```

Override.
```
public LTS<S, T> clone();
```

Override
```
public String int toString();
```

Generate a ".dot" file for the LTS.
Pre: filename != null
Post: A dot file for this LTS has been generated and stored in the file at the specified location
```
public void saveAsDotFile (String filename);
```

A.3.6 LTSLoader

Methods

Load the LTS from the specified file
Pre: filename != null
Post: The returned value contains the LTS represented in filename if the file exists, null otherwise
```
public static LTS<StateImpl,TransitionImpl> loadLTS (String filename);
```

Save the lts as a ".lts" file.
Pre: filename, lts != null
Post: lts has been save in filename.
```
public static void saveLTS (LTS<StateImpl,TransitionImpl> lts, String filename);
```

A.4 Inner classes of LTS

A.4.1 InnerState<S extends StateAbstract, T extends TransitionAbstract>

Fields

```
private final S state;
private final HashSet<T> in;
private final HashSet<T> out;
```

Constructors

Pre: s != null
Post: An instance of this is created, representing an internal state with specified S state
```
public InnerState (S s);
```

Methods

Get the ingoing transitions.
Pre: -
Post: The returned value contains the set of transitions for which this is the destination state
```
public HashSet<T> getInTransitions();
```

Get the outgoing transitions.
Pre: -
Post: The returned value contains the set of transitions for which this is the source state
```
public HashSet<T> getOutTransitions();
```

Add an ingoing transition to the state.
Pre: newTransition != null
Post: The specified transition has been added at the set of ingoing transitions of this
`public void addInTransition (T newTransition);`

Add an outgoing transition to the state.
Pre: newTransition != null
Post: The specified transition has been added at the set of outgoing transitions of this
`public void addOutTransition (T newTransition);`

Remove delT from the ingoing transitions to this.
Pre: delT != null
Post: The specified transition has been removed at the set of ingoing transitions of this
`public void removeInTransition (T delT);`

Remove delT from the outgoing transitions to this.
Pre: delT != null
Post: The specified transition has been removed at the set of outgoing transitions of this
`public void removeOutTransition (T delT);`

Override
`public String int toString();`

A.4.2 `InnerTransition<S extends StateAbstract, T extends TransitionAbstract>`

Fields

```
private final T transition;
private final S from;
private final S to;
```

Constructors

Pre: t, from, to != null
Post: An instance of this is created, representing an internal transition with specified corresponding transition, source and destination.
`public InnerTransition (T t, S from, S to);`

Methods

Get the the source state of this.
Pre: -
Post: The returned value contains the the source state of this.
`public S getSource();`

Get the the destination state of this.
Pre: -
Post: The returned value contains the the destination state of this.
```
public S getDestination();
```

Get the corresponding transition.
Pre: -
Post: The returned value contains the corresponding transition.
```
public T getTransition();
```

Override
```
public String int toString();
```

APPENDIX

B

API OF THE MUT PACKAGE

B.1 Interface

B.1.1 MutOpIF<S extends StateAbstract, T extends TransitionAbstract>

Methods

Get a new random mutant not yet generated by the operator
Pre:-
Post: The returned value contains a new random mutant not yet generated by the operator
```
public LTS<S, T> mutate();
```

Get the number of mutants possible to generate
Pre:-
Post: The returned value contains the number of mutants possible to generate
```
public int getMaxMutants();
```

Get the name of the mutation operator
Pre:-
Post: The returned value contains the name of the mutation operator
```
public String getName();
```

B.2 Abstract classes

B.2.1 MutOpAbsract<S extends StateAbstract, T extends TransitionAbstract>

implements MutOpIF<S, T>

Fields

```
protected String name;
protected int acc;
```

```
protected int maxMutants;
protected int maxMutateCalls;
protected LTS<S, T> toMutate;
```

Methods

Overrides
```
public int getMaxMutants();
```

Overrides
```
public String getName();
```

B.3 Classes

Since the implementation of each mutation operator are very similar, only the API of COL (change of label) is provided. The other mutation operators implemented are :

- `IOT` (insertion of transition)
- `MOT` (missing of transition)
- `MOT2`
- `MOT3`
- `ROT` (reverse of transition)
- `SSC` (starting state changed)

For more information about those implementations, read Section 4.2

B.3.1 `COL`

extends MutOpAbsract<StateImpl, TransitionImpl>

Fields

```
private Map<Integer, boolean[]> done;
private LabelIF[] labels;
private List<TransitionImpl> transitions;
private List<StateImpl> from;
private List<StateImpl> to;
```

Constructor

Pre: toMutate != null
Post: An instance of this is created, representing the mutation operator COL with specified model on which mutate.
`public COL(LTS<StateImpl, TransitionImpl> toMutate);`

Methods

Overrides
`public LTS<StateImpl, TransitionImpl> mutate();`

Get a new random mutant not yet generated
Pre: i, j, labelAcc ≥ 0
Post: The returned value contains a new random mutant not yet generated.
`public LTS<StateImpl, TransitionImpl> mutate(int i, int j, int labelAcc);`

B.3.2 `Pair`

Fields

```
private final int a;
private final int b;
```

Constructor

Pre: -
Post: An instance of this is created, representing a pair of integers.
`public Pair(int a, int b);`

Methods

public int getA() return a; /** * @return the b */ public int getB() return b;
Get the element a of the pair
Pre:-
Post: The returned value contains the first element of the pair
`public int getA();`

Get the element b of the pair
Pre:-
Post: The returned value contains the second element of the pair
`public int getB();`

Override
```
public abstract boolean equals (Object o);
```

Override
```
public abstract int hashCode();
```

APPENDIX

C

API OF THE CHECKPROP PACKAGE

For more information about those implementations, read Section 4.3

C.1 Classes

C.1.1 Checker<S extends StateAbstract, T extends TransitionAbstract>

Fields

```
private final LTS<S, T> mental;
private final LTS<S, T> system;
private LTS<CompState<S>, TransitionImpl> origComp;
private LTS<CompState<S>, TransitionImpl> lastComp;
private int kills = 0;
```

Constructor

Pre: system, mental != null
Post: An instance of this is created, representing a property checker with specified system model and mental model
`public Checker(LTS<S, T> system, LTS<S, T> mental);`

Methods

Check if the mutant kills involuntarily
Pre:-
Post: The returned value contains true if the composition reaches a state which contains
"spreader=OUT_PLACE,
nbeamFire=FIRED,
nbeamLevel=X_SET" in its name and false otherwise.
`public boolean isKilling();`

Get an LTS representing the composition between mental and system
Pre:-
Post: The returned value contains an LTS representing the composition between mental and system.
```
public LTS<CompState<S>, TransitionImpl> getOrigComp();
```

Get the last computed composition
Pre:-
Post: The returned value contains the last computed composition.
```
public LTS<CompState<S>, TransitionImpl> getLastComp();
```

Check if system is fully controllable by mental and build the composition
Pre: mutant != null
Post: The returned value contains true if system is fully controllable by mental, false otherwise.
```
public boolean isFullControl ();
```

Check if system is fully controllable by mutant and build the composition
Pre: mutant != null
Post: The returned value contains true if system is fully controllable by mutant, false otherwise.
```
public boolean isFullControl (LTS<S, T> mutant);
```

Check if system is "alternatively fully controllable" by mutant and build the composition
Pre:-
Post: The returned value contains true if system is "alternatively fully controllable" by mutant, false otherwise.
```
public boolean isAlternativeFullControl (LTS<S, T> mutant);
```

Fill a table of statistics with the errors with respect to full-control
Pre: mutant != null
Post: The returned value contains the entries of the table of statistics
```
public Set<StatTableEntry<S» statFullControl(LTS<S,T> mutant);
```

Fill a table of statistics with the errors wrt full-control but stop the traversal further a mode confusing state.
Pre: mutant != null
Post: The returned value contains the entries of the table of statistics
```
public Set<StatTableEntry<S» statFullControlMode(LTS<S,T> mutant);
```

C.1.2 `StatTableEntry<S extends StateAbstract>`

Fields

```
private CompState<S> failState;
private Collection<LabelIF> missingCmds;
```

```
private Collection<LabelIF> extraCmds;
private Collection<LabelIF> missingObs;
private double percMisCmds;
private double percMisObs;
```

Constructor

Pre: failState != null
Post: An instance of this is created, representing an entry in a table of statistics with specified related composite state.
`public StatTableEntry(CompState<S> failState);`

Pre: failState, missingCmds, extraCmds, missingObs != null
Post: An instance of this is created, representing an entry in a table of statistics with specified related composite state, and the set of missing commands, extra commands, missing observations.
`public StatTableEntry(CompState<S> failState, Collection<LabelIF> missingCmds, Collection<LabelIF> extraCmds, Collection<LabelIF> missingObs);`

Methods

Get the percentage of missing commands
Pre:-
Post: The returned value contains the percentage of missing commands
`public double getPercMisCmds();`

Set the percentage of missing commands
Pre: percMisCmds != null
Post: percMisCmds is set to the specified value
`public void setPercMisCmds(double percMisCmds);`

Get the percentage of missing observations
Pre:-
Post: The returned value contains the percentage of missing observations `public double getPercMisObs();`

Set the percentage of missing observations
Pre: percMisObs != null
Post: percMisObs is set to the specified value
`public void setPercMisObs(double percMisObs);`

Add a missing command.
Pre: toAdd != null
Post: toAdd has been added to the set of missing commands
`public void addMissingCmd(LabelIF toAdd);`

Add an extra command.
Pre: toAdd != null
Post: toAdd has been added to the set of extra commands
`public void addExtraCmd(LabelIF toAdd);`

Add a missing observation.
Pre: toAdd != null
Post: toAdd has been added to the set of missing observations
`public void addMissingObs(LabelIF toAdd);`

Override
`public String int toString();`