## Contents

1 Introduction .......................... 5
   1.1 The problem .......................... 5
   1.2 An overview of the solution .................. 5
   1.3 Document Overview ........................ 6

2 Theoretical Background ................. 7
   2.1 The software testing process ............... 7
      2.1.1 Testing and software verification and validation 7
      2.1.2 Testing in system engineering ............. 7
      2.1.3 Testing - concepts and definitions ........ 9
   2.2 Debugging ............................. 11
      2.2.1 The scientific method .................. 11
      2.2.2 Deriving a hypothesis .................. 12
      2.2.3 Reasoning techniques in debugging ........ 13
   2.3 Fault models for object-oriented programs .... 14
   2.4 Design By Contract ..................... 15
      2.4.1 Overview ............................ 15
      2.4.2 Definitions and concepts ............... 16
      2.4.3 Benefits of assertions ................ 18
      2.4.4 Languages implementing Design by Contract 18
   2.5 Testing based on Design by Contract ......... 19
   2.6 Eiffel - brief overview of the language .... 20
      2.6.1 Concepts ............................ 20
      2.6.2 Eiffel Syntax ........................ 20

3 Related work .......................... 23
   3.1 Testing strategies and testing tools ........ 23
   3.2 Bug classification ..................... 25
4 Classes of Bugs in Eiffel Code

4.1 The testing process ............................................. 26
4.2 The debugging process ......................................... 26
4.3 Types of faults in Eiffel ....................................... 28
   4.3.1 Implementation/Specification ............................ 28
   4.3.2 Supplier-induced bug ..................................... 38
   4.3.3 Inheritance ................................................ 43
   4.3.4 Special cases .............................................. 48

5 AutoTest .............................................................. 54

5.1 Installation .......................................................... 54
5.2 Command line arguments ....................................... 54
5.3 Ace files ............................................................ 55
5.4 AutoTest - behind the scenes .................................. 57
5.5 Test execution: the driver and the interpreter ............... 59
   5.5.1 The interpreter ............................................. 59
5.6 AutoTest Output .................................................... 60

6 Implementation ...................................................... 61

6.1 Testing a routine of a class ...................................... 61
6.2 System Design ..................................................... 61
   6.2.1 Process the input arguments .............................. 62
   6.2.2 Compilation ................................................ 62
   6.2.3 Generation of the interpreter ............................. 64
   6.2.4 Execute tests .............................................. 64
   6.2.5 The result repository .................................... 64
   6.2.6 Statistics ................................................... 64
   6.2.7 Particular routine testing ................................ 65
6.3 Bug report tool ..................................................... 68
   6.3.1 Technology ................................................ 68
   6.3.2 Design ....................................................... 69
   6.3.3 User Interface Functionality .............................. 71
   6.3.4 The process management layer ......................... 74
   6.3.5 The storage layer ......................................... 75
List of Figures

1.1 Fault classification in automatic testing .......................... 6
2.1 A systems engineering view of testing ............................ 8
2.2 Testing and its results in the software development process .... 9
2.3 A hierarchy of program analysis techniques ....................... 14
4.1 Debugging in Eiffel Studio ........................................ 27
4.2 Analysis and implementation - schematic view .................... 28
4.3 Implementation and specification bugs ........................... 29
4.4 Inheritance hierarchy for routine append in LINKED_QUEUE ... 45
5.1 An example of ace file .............................................. 56
5.2 AutoTest html results .............................................. 60
6.1 AutoTest Classes .................................................. 63
6.2 Class diagram for the auto_test cluster ......................... 65
6.3 Bug Report Tool - class diagram ................................ 70
6.4 Bug Report Tool - class diagram ................................ 71
6.5 Interface of the bug report tool .................................. 72
6.6 Additional error/warning-message dialogs ....................... 73
7.1 Distribution of bugs in EiffelBase ................................. 77
Abstract

Performing random contract-based testing on widely-used libraries and production software is an important task in the software development process. AutoTest has been used intensively for testing EiffelBase, Gobo and other software written in Eiffel. A close observation of the faults in the tested code, supported by a debugging process lead to a catalog of bugs. This document focuses on the categories of faults found by AutoTest in different Eiffel applications, shows example and possible solutions for solving faulty situations.
1 Introduction

1.1 The problem

An important step in the software development process is testing. Each deployed component should be pass a testing phase in order to discover possible faults and errors. Automatic testing cannot cover all the input combinations of arguments, but it can generate cases that the designer of the component has not thought of.

The testing process can be time consuming, tiresome and even boring sometimes. There are situations in which the coder does not have enough resources (in space and time) for performing sufficient testing on his/her code. This is why several tools for automatic testing can be used. Some of these existing tools do their job pretty well, and some even provide a list of test cases that generate bugs, but they do not cover the debugging and interpretation process required for building a bug object model, for developing a category of faults.

1.2 An overview of the solution

This diploma project focuses on reporting and analysing the faults in Eiffel code. We have made a large field case study (at the level of hundreds of classes tested). By observing the conditions under which a bug appears we have created a classification scheme of the bugs in Eiffel. Figure 1.1 shows where this classification comes into scene when testing a software system.

In order to speed up the activity of creating a bug report I have build a tool that provides interfaces for creating a cluster detailed report that contains class information and for each class routine bug data.

The inheritance problems that appeared during the testing process led to the idea of modifying the existing tool, AutoTest in order to accept as parameter just a set of features/routines.
1 Introduction

Figure 1.1: Fault classification in automatic testing

1.3 Document Overview

Chapter 2 – Theoretical Background provides the basic information for understanding the concepts that have been used for developing the classification scheme. Mainly I will present the software testing and debugging processes, I will give a short introduction to fault models in object-oriented systems and I will explain briefly the ideas behind Design by Contract™. As the whole diagnosis is done for Eiffel, I will also present a short summary of this programming language.

Chapter 3 – Related work positions the tool I have used between other testing tools in what regards the testing strategy. It also presents other bug classification attempts for Eiffel code.

Chapter 4 – Classes of Bugs in Eiffel Code is the main part of my research work. I will describe the classification process and the steps that lead to the existing categories of bugs. Each category will be detailed and exemplified.

Chapter 5 – AutoTest presents the structure and functioning of AutoTest.

Chapter 6 – Implementation presents the structure of the cluster created for testing just some routines of a class and the design and architecture of the graphical user interface for the bug classification procedure.

Chapter 7 – Results will show the test statistics on the libraries and applications on which AutoTest was run.

Chapter 8 – Directions for future work gives an idea for a future integration of the GUI to AutoTest.

Chapter 9 – Conclusions gives the main ideas and issues that were concluded from the worked carried out.
2 Theoretical Background

2.1 The software testing process

One of the major steps in the development of a software product is testing. Software testing is the execution of code using combinations of input and state selected to reveal bugs.[3]

2.1.1 Testing and software verification and validation

Testing is part of the verification and validation phase that should be carried out through the whole software development process. Verification is concerned with the evaluation of the product during each step of its life cycle in order to ensure that it meets the requirements of the previous step. It tries to answer the question “Am I building the product right?” . On the other hand, validation determines if the system complies with the requirements, if it performs functions for which it is intended and if it meets the goals of the users and of the stakeholders. It tries to answer the question “Am I building the right product?”. “It usually relies on a combination of reasoning and testing, and encompasses unit, integration, acceptance testing.”[9]

2.1.2 Testing in system engineering

We will go deeper and try to present testing as a problem in systems engineering. As it is emphasised in [3], it is the design and implementation of a special kind of software system: one that exercises another software system with the intent of finding bugs. Tests are designed by analysing the system under test and deciding how it is likely to be buggy. In turn, test design provides the requirements for the test automation system. This system automatically applies and evaluates the tests. It must be designed to work with the physical interfaces, structure and the runtime environment of the system under test. Manual testing, of course still plays a role. But testing is mainly about the development of an automated system to implement an application specific test design.
The design of a test automation is a complex process. A schematic view is given in Figure 2.1 as presented in [3].

Test design requires solving problems similar to those encountered in the analysis, design and programming of an application system. Test models are developed to represent responsibilities. Test design involves several steps:

1. Identify, model and analyze the responsibilities of the system under test
2. Design test cases based on this external perspective
3. Add test cases based on code analysis, suspicions and heuristics
4. Develop expected results for each test case or choose an approach to evaluate the pass/no pass status of each test case

After design is complete, the tests are applied to the system under test. Unless manual testing is indicated, a test automation system must be developed to run the tests. A test automation system typically will start the implementation under test, set up its environment, bring it to the required pretest state, apply the test inputs and evaluate the resulting output and state.
Test execution typically follows several steps:

1. Establish that the implementation under test is minimally operational by exercising the interfaces between its parts.

2. Execute the test suite; the result of each test is evaluated as pass or no pass.

3. Use a coverage tool to instrument the implementation under test.

4. If necessary, develop additional tests to exercise uncovered code.

5. Stop testing when the coverage goal is met and all tests pass.

Testing design and execution are carried out in parallel with application analysis, design and implementation as depicted in Figure 2.2.

![Figure 2.2: Testing and its results in the software development process](image)

2.1.3 Testing - concepts and definitions

Next a series of definitions will be provided. They will help the reader understand the notions presented further in the project. All the ideas are summarized from [3].
Software testing is the execution of code using combinations of input and state selected to reveal bugs.

Automated testing: software testing assisted with software tools that require no operator input, analysis, or evaluation.

Manual testing: that part of software testing that requires operator input, analysis, or evaluation.

The scope of a test is the collection of software components to be verified. A component refers to any software aggregate that has visibility in a development environment. The code being tested is called the implementation under test (IUT), method under test (MUT), object under test (OUT), class under test, component under test, system under test (SUT) and so on. Scope is traditionally designated as unit, integration or system. The scope of a unit test comprises a small executable (in object oriented programming, an object of a class is the smallest executable unit). The scope of an integration test is a complete system or subsystem of software and hardware units. The units included are physically dependent or must cooperate to meet some requirement. The scope of a system test is a complete integrated application.

A test case specifies the pretest state of the IUT and its environment, the test inputs or conditions, and the expected results.

A test suite is a collection of test cases, typically related by testing goal or implementation dependency.

A test oracle is a source of expected results for a test case.

A failure is the manifested inability of a system or component to perform a required function within the specified limits. It is evidenced by incorrect output, abnormal termination, or unmet time and space constraints.

A software fault is missing or incorrect code. When the executable code (translated from faulty statements) is executed it may result in a failure. An error is a human action that produces a software fault. Errors, faults and failures do not occur in one-to-one relation. That is, many different failures can result from a single fault, and the same failure can be caused by different faults. Similarly, a single error may lead to many faults.

A bug is an error or a fault.
2.2 Debugging

The process of debugging can be defined as the work required to diagnose and correct a bug. *Testing is not debugging. Debugging is not testing.*

Debugging typically occurs after a failure has been observed or when a developer identifies a fault by inspection or automatic code analysis. It includes the analysis and experimentation necessary to isolate and diagnose the cause of failure, the programming to correct the bug, and testing to verify that the change has removed the bug. The observed failure may result from testing (which is desirable) or from the intended use of a system (which is undesirable). In contrast, testing is the process of devising and executing a test suite that attempts to cause failures.

Once the failure has been produced, one should understand how it came to be. The whole work carried out for this project is based on a complex process of debugging that relies on the *scientific method* introduced by \[18\]. This method is an appropriate process for obtaining problem diagnostics as it does not require a priori knowledge and works in a systematic and reproducible fashion such that one can be sure to eventually find the cause and reproduce it at will.

2.2.1 The scientific method

The scientific method approaches the failing program as if it were a natural phenomenon. Its main steps are:

1. Observe a failure (i.e. as described in the problem description)
2. Invent a tentative description, called *hypothesis* as to the failure cause that is consistent with the observations.
3. Use the hypothesis to make *predictions*.
4. Test the hypothesis by *experiments* or further *observations*:
   - If the experiment satisfies the predictions, refine the hypothesis
   - If the experiment does not satisfy the predictions, create an alternate hypothesis
5. Repeat steps 3 and 4 until the hypothesis can no longer be refined.

When all the discrepancies are gone, the hypothesis becomes a *theory* about how the failure came to be:
2 Theoretical Background

• It explains earlier observations (including the failure)

• It predicts future observations (for instance, that the failure no longer appears after applying a fix).

In the context of fault identification and classification, such a theory is called a diagnosis.

2.2.2 Deriving a hypothesis

The scientific method gives a general process for turning a hypothesis into a theory - or more specifically, an initial guess into a diagnosis. But, in each iteration of the method one should come up with a new hypothesis. This is actually the creative part of debugging: thinking about the many ways a failure could have come to be. Being creative is not enough. One should also be effective in the action he/she is doing. The better the hypotheses the less iterations should be completed and the diagnosis is faster. In order to be effective, one needs to leverage as many knowledge sources as possible. I will present them shortly, as described in [18].

The description of the problem: Without a concise description of the problem one will not be able to tell if the problem is solved or not. A simplified problem report is useful for this issue.

The program code: is the common abstraction across all possible program runs, including the failing run. It is the basis for almost all debugging techniques. Without knowledge about the internals of the program, one can only observe concrete runs (if any) without ever referring to the common abstraction. Lack of program code makes understanding (and thus debugging) much more difficult.

The failing run: the program code allows one to speculate about what may be going on in a concrete failing run. If one actually executes the program such that the problem is reproduced, one can observe actual facts about the concrete run. Such facts include the code being executed and the program state as it evolves. These observation techniques are the bread and butter of debugging.
Alternate runs: a single run of a nontrivial program contains a great deal of information, and thus one needs a means of focusing on specific aspects of the execution. In debugging, one is most interested in anomalies - those aspects of the failing run that differ from "normal" passing runs. For this purpose one should know which "normal" runs exist, what are their common features, and how these differ in the failing run.

Earlier hypothesis: depending on the outcome of a scientific method experiment, one must either refine or reject a hypothesis. In fact, every new hypothesis must:

- include all earlier hypotheses that passed (whose predictions were satisfied) and
- exclude all hypotheses that failed (whose predictions were not satisfied).

Any new hypothesis must also explain all earlier observations, regardless of whether the experiment succeeded or failed - and it should be different enough from earlier hypotheses to quickly advance toward the target.

2.2.3 Reasoning techniques in debugging

Depending on the ingredients that make up a certain situation, humans use different reasoning strategies to learn about programs. These techniques are described in [18]. I will present them briefly. Figure 2.3 shows the hierarchy formed by these strategies.

Deduction: is reasoning from the general to the particular. It lies at the core of all reasoning techniques. In program analysis, deduction is used for reasoning from the program code (or other abstractions) to concrete runs - especially for deducing what can or cannot happen. These deductions take the form of mathematical proofs.

Observation: allows the programmer to inspect arbitrary aspects of an individual program run. It brings in actual facts of a program execution.

Induction: is reasoning from the particular to the general. In program analysis, induction is used to summarize multiple program runs (e.g. a test suite or random testing) to some abstraction that holds for all considered program runs.
Experimentation: searching for the cause of a failure using the scientific method requires a series of experiments, refining and rejecting hypotheses until a precise diagnosis is isolated. This implies multiple program runs that are controlled by the reasoning process.

2.3 Fault models for object-oriented programs

Testing presents fundamental and interesting situations. A basic question would be: what is the input, path or state of a system under test that should be tried? When should the test be stopped? If one relies on testing to prevent certain kinds of failures, how can one design systems that are both testable and efficient? How can one design a test suite that is small enough to be practical and in the same time ‘big enough’ to demonstrate failure-free operation?

As it is underlined in [3], while one can be certain there are unknown bugs in any nontrivial software system, one does never know exactly where they are (otherwise, there would be no need to test). The number of places to look is infinite for practical purposes, so any rational testing strategy must be guided by a fault model. A fault model answers a simple question about a test technique: Why do the features called out by the technique warrant our effort. This operational definition identifies relationships and components
of the system under test that are most likely to have faults. The answer to the question may be based on common sense, experience, suspicion, analysis or experiment.

A **bug hazard** is a circumstance that increases the chance of a bug. Such hazards arise for many reasons in many situations. Once identified, a bug hazard provides the basis for a fault model.

Software testing strategies are effective to the extent that their fault model is a good predictor of faults. The choice of a test model suggests a **test strategy**. A test strategy yields a test suite when applied to a representation or implementation.

The object oriented programming paradigm presents a unique blend of powerful constructs, bug hazards and testing problems. This is an unavoidable result of the encapsulation of operations and variables into a class, the variety of ways a system of objects can be composed, and the compression of complex runtime behavior into a few simple statements. For example each lower level in an inheritance hierarchy creates a new context for inherited features; correct behavior at an upper level in no way guarantees correct behavior at a lower level.

### 2.4 Design By Contract

#### 2.4.1 Overview

Design by contract (DBC), or Programming by contract is a methodology for designing computer software. It is a trademark of Eiffel Software Inc., the designers of Eiffel. The basic idea behind it is to view the relationship between a class and its clients as a formal agreement, expressing each party’s rights and obligations. It is derived from the correctness formulas or Hoare triples used in formal program verification [13]:

Let $A$ be some operation (for example an instruction or a routine body).

A **correctness formula** is an expression of the form

$$\{P\}A\{Q\}$$

denoting the following property which may or may not hold:

Meaning of a correctness formula $\{P\}A\{Q\}$

“Any execution of $A$, starting in a state where $P$ holds, will terminate in a state where $Q$ holds.”

In $\{P\}A\{Q\}$: $A$ denotes an operation; $P$ and $Q$ are properties of the various entities involved, also called assertions. $P$ is called the precondition and $Q$ is the postcondition.
One could look at assertions as containing the specification of a software system. So, the correctness of a software element can be defined as the consistency of its implementation with its specification.

### 2.4.2 Definitions and concepts

#### Assertions

In order to express a specification in Eiffel one relies on assertions. An assertion is an expression involving some entities of the software, and stating a property that these entities may satisfy at certain stages of software execution. A typical assertion (as illustrated in the chapter *Classes of bugs in Eiffel Code*) might express that a certain integer has a positive value or that a certain reference is not void etc.

Syntactically, in Eiffel assertions are boolean expressions or they may include the `old` keyword in postconditions.

#### Preconditions and postconditions

As underlined in [13], the one of the uses of assertions is the semantic specification of routines (a routine in Eiffel is the similar of a Java *method*). A routine is not just a piece of code, it is a feature that should perform a useful task.

One may specify the task performed by a routine by two assertions associated with the routine: a *precondition* and a *postcondition*.

The precondition states the properties that must hold whenever the routine is called. (it is a boolean expression that must be true before every call to the feature).

The postcondition states the properties that the routine guarantees when it returns (it is a boolean expression which must hold after every call to the feature, and can include the `old` keyword).

It is of use to emphasize some properties of assertions for a good understanding and usage:

- Assertions are not an input checking mechanism. A precondition will not take care of correcting user input for example.

- Assertions are not control structures. They express correctness conditions and do not represent techniques for handling special situations.
Next I will present two assertion violation rules described largely in [13]. These rules represent the basis of my fault classification system.

*Assertion violation rule (1):* A run-time assertion violation is the manifestation of a bug in the software.

*Assertion violation rule (2):*  
A precondition violation is the manifestation of a bug in the client.  
A postcondition violation is the manifestation of a bug in the supplier.

A precondition violation means that the routine’s caller, although obligated by the contract to satisfy a certain requirement, did not. This is a bug in the client itself.

A postcondition violation means that the routine, presumably called under correct conditions, was not able to fulfill its contract. In this case the bug is in the routine and the caller is not faulty.

**Clients and suppliers**

Let S be a class. A class C which contains a declaration of the form a: S is said to be a client of S. S is then said to be a supplier of C. [13]

**Class invariants**

Preconditions and postconditions describe the properties of individual routines. The global properties of the instances of a class should also be expressed. They must be preserved by all routines. The class invariant contains such properties and captures the deeper semantic properties and integrity constraints characterizing a class.

An invariant of a class C is a set of assertions that every instance of C will satisfy at all “stable” times. Stable times are those in which the instance is in an observable state:

- On instance creation (after execution of create), where a is of type C)
- Before and after every call a.r(...) to a routine a of the class.

**Check instruction**

The check instruction is a boolean expression which must hold at a certain position in a statement block.
Loop invariants and variants

The loop invariant and variant help the coder guarantee that a loop is correct.

A loop invariant is a boolean expression which must hold at the beginning and the end of each loop iteration. [11]

A loop variant is an integer expression which decreases with every loop iteration.

2.4.3 Benefits of assertions

Assertions provide the following advantages:

- they can be used as a tool for writing correct software
- assertions are part of the source code and hence form a good documentation.
- enabling assertion checking during runtime reveals bugs earlier and closer to the source of the bug.

2.4.4 Languages implementing Design by Contract

There are several languages that implement the concept of design by contract. I will provide a short list for the reader to make an opinion on the variety of languages this concept covers.

- Eiffel
- C (a preprocessor written in Ruby)
- C++ (the tool C2)
- C# (tools: eXtensible C#, Spec#)
- Chrome
- D (implements DBC as a major feature)
- Java (tools: iContract2, Contract4J, jContractor, Jcontract, C4J)
- JML (provides specifications, including contracts, to describe Java programs)
• Lisaac (implements DBC as a major feature)
• Perl (with the Class::Contract module)
• Perfect (with the tool Perfect Developer implements an extension of DBC called Verified Design by Contract)
• PHP (implements DBC via its \texttt{assert()} function and a callback function defined using \texttt{assert_options()})
• Python (tools that support DBC are PyDBC, Contracts for Python)
• SPARK (implements DBC by static analysis of Ada programs)

\section*{2.5 Testing based on Design by Contract}

Eiffel code is equipped with assertions. Starting from this idea, one should be able to prove mathematically, that the routine implementations are consistent with the assertions.

As long as one knows what the software should do, there is no need for human intervention in order to test it. In Eiffel, the contracts (assertions) contain the specification of the software. This lead to the idea of \textit{contract-based testing} i.e. using contracts as defined in Design by Contract$^\text{TM}$ as an automatic, freely-available testing oracle and for input value generation.

Contracts contain the intended semantics of the software, they are integrated with the source code, and are executable. Hence they can be used as an oracle (the part of the testing system that decides if a test case has passed or failed).

The tool that I have used is AutoTest developed by Ilinca Ciupa and Andreas Leitner (available at http://se.ethz.ch/people/leitner/autotest). AutoTest is a fully automatic testing tool. The information that contracts (preconditions, postconditions, class invariants, loop variants and invariants, and check instructions) provide is used to check whether the software fulfills its intended purpose. By checking that the software respects its contracts, one can ascertain its validity. Therefore, contracts provide the basis for automation of the testing process.

AutoTest allows the user to generate, compile and run tests on the push of a button.
2.6 Eiffel - brief overview of the language

Eiffel is a strongly typed, pure object-oriented language.

2.6.1 Concepts

As described in [13], object orientation is primarily an architectural technique: its major effect is on the modular structure of software systems.

The key role is played by classes. A class describes not just a type of objects but also a modular unit. In a pure object-oriented approach, *classes should be the only modules.*

In particular, there is no notion of main program, and subprograms do not exist as independent modular units. They may only appear as part of classes. For ease of management, classes are grouped into administrative units, called *clusters.*

The notion of class is powerful enough to avoid the need for any other typing mechanism. *Every type should be based on a class.* Feature call is the primary computational mechanism. It is possible for the author of a class to specify that a feature is available to all clients, no no client, or to specified clients.

The language provides a mechanism to recover from unexpected abnormal situations (exception handling).

It is possible to write classes with formal generic parameters representing arbitrary types.

Single and multiple inheritance mechanisms are available. A class may inherit from as many others as necessary, with an adequate mechanism for disambiguating name clashes. Precise rules govern the fate of features under repeated inheritance allowing developers to choose separately for each repeatedly inherited feature, between sharing and replication. The specification, signature and implementation of an inherited feature can be redefined.

2.6.2 Eiffel Syntax

Responsibility assertions are an integral part of class definitions in Eiffel. Class and routine (method) declarations may include assertions. A class declaration is given below:

```
class <class declaration>

feature
  <routine interface declaration> is
    require <precondition predicate>
```
local <local_declarations>
do <routine_body>
ensure <postcondition predicate>
rescue <exception expression>
end

feature ...

invariant <invariant_predicate>
end

Table 2.1 depicts the semantics of the assertion-related keyword expressions (as presented in [3]).

Eiffel supports build-in loop assertions. The check <predicate> expression can be used at any point in the code. Compilation parameters can enable/disable assertions. All assertions use the same runtime mechanism. When an enabled assertion is reached, the predicate is evaluated for the current contents of the object. If the result is false, an assertion exception is raised. If the result is true execution continues.
## 2 Theoretical Background

<table>
<thead>
<tr>
<th>Assertion Type/Semantics</th>
<th>Eiffel Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class invariant</td>
<td><code>invariant</code> predicate</td>
</tr>
<tr>
<td>Precondition</td>
<td><code>require</code> predicate</td>
</tr>
<tr>
<td>Postcondition</td>
<td><code>ensure</code> predicate</td>
</tr>
<tr>
<td>Weakened precondition of a redeclared routine (overridden method)</td>
<td><code>require else &lt;predicate&gt;</code></td>
</tr>
<tr>
<td>Strengthened postcondition of a redeclared routine (overridden method)</td>
<td><code>ensure then &lt;predicate&gt;</code></td>
</tr>
<tr>
<td>General purpose</td>
<td><code>check &lt;predicate&gt;</code></td>
</tr>
<tr>
<td>Value of <code>&lt;entityName&gt;</code> at entry. Typically used in postcondition predicates</td>
<td><code>old &lt;entityName&gt;</code></td>
</tr>
<tr>
<td>Assertion action. <code>&lt;expression&gt;</code> specifies how the Eiffel exception mechanism will be used.</td>
<td><code>rescue &lt;expression&gt;</code></td>
</tr>
<tr>
<td>Loop invariant; variant</td>
<td><code>from &lt;initialization expression&gt;</code> <code>invariant &lt;predicate&gt;</code> <code>variant &lt;predicate&gt;</code> <code>until &lt;predicate&gt;</code> <code>loop &lt;loop body&gt;</code> <code>end</code></td>
</tr>
</tbody>
</table>

Table 2.1: Elements of Eiffel Assertion Checking
3 Related work

3.1 Testing strategies and testing tools

As presented in [6] AutoTest:

- It is completely automatic (requires no intervention of the user whatsoever).
- It has been applied to the programming methodology which introduced contracts in the software development cycle (Eiffel).
- It has been tested on full-fledged, industrial-scale applications.

The target of AutoTest is to develop a complete process for automatic testing, not just one or several of its separate stages. It is with regard to this that the ideas behind AutoTest are fundamentally different from other approaches, which focus on automating only certain aspects and do not deal at all with the others.

The results published in several papers support the idea of using contracts for testing. [5] tries to estimate the improvement in the fault isolation effort that contracts bring. The authors use the notion of diagnosability to illustrate this concept: “diagnosability captures the fault isolation effort after the occurrence of a failure”. The results presented in the paper show that diagnosability improves considerably (about 8 times) in the presence of contracts defined with very high precision. [16] investigates the question of which types of assertions are most effective at detecting faults. The author develops a classification of assertions from this perspective. This classification is meant as a first step in creating a methodology of programming with assertions.

Researchers have dedicated a large amount of work to automatic testing. The idea of using contracts as built-in test (as discussed in [3]) has been exploited as part of several approaches. [4] presents a method that is very similar to that of AutoTest: the testing tool called Korat uses preconditions to generate valid and non-isomorphic inputs for each method under test and postconditions to evaluate the correctness of the methods. They place special emphasis on the technique of generating inputs based on
3 Related work

parsing the method preconditions and on a user-provided finitization function on the inputs. Korat can automatically generate test cases for Java programs with associated JML [10] assertions. The disadvantage of this approach is that it is not fully automatic: although a skeleton for the finitization function is generated by the tool, in many cases users are required to edit it to get the behavior they want. Various research groups have investigated the idea of using specifications for testing. [15] is the original paper formalizing specification-based testing. They propose the extension of structural testing with specification-based techniques. Test cases contain preconditions and postconditions. Two of the approaches they develop (specification/error-based testing and specification/fault-based testing) assume that the specification itself may be faulty. The other two approaches (oracle/error-based testing and oracle/fault-based testing) treat the specification as an oracle, as is the case in our work. However, this is the only similarity between the work described in [15] and AutoTest, as they only address the issue of automating the test oracle. [12] also uses method preconditions and postconditions to model the functional correctness of systems. The author views testing as a constraint solving or satisfiability problem: determining values for the input variables so that the system under test terminates and the conjunction between the precondition and the negated postcondition is satisfied. The approach is based on building an approximate model of the system by running several tests on it and then using this knowledge to estimate where a successful test case might be found. The idea is interesting, but evaluation of the performance of the algorithm was still in progress when the paper was written. The author provides only a simple case study, from which it is difficult to make any estimation about the general performance of the solution.

There are several testing tools on Eiffel code. In the eiffel-zone web location (http://eiffelzone.com/esd/tool-t.html) one can find 11 testing packages.

The most important are:

- AutoTest - largely described in the implementation section of this document
- Swete - a cross platform console based regression test tool for web applications written in Eiffel.
- Eiffel Spec Test - ES-Test (formerly ETest) is a unit-testing framework for Eiffel. Test driven development is a three-step process: write a unit test, write code, refactor.
- Gobo Eiffel - is a blending of tools and libraries that provide support for writing
3 Related work

Eiffel applications. It contains libraries for compilers, data structures, date and time support, files and directories, design patterns, lexical analysis, xml.

- eWeasel - was used for testing the Eiffel compiler (available with the sources of EiffelStudio)

3.2 Bug classification

Bug classification in Eiffel is a completely new topic in the field. This is the main subject of my diploma thesis.

An attempt of providing classes of bugs has been done in [3], but with a different approach. The book introduces the test design pattern presented at three main scopes: class, subsystem and application system. The pattern template focuses on the following dimensions of the test design:

- When is a particular test strategy appropriate?
- What kind of bugs will it find?
- How do you develop a test suite – how should the implementation under test be modeled, and how are test cases produced from the model and its oracle?
- What kind of test automation works best?
- What are the testing entry and exit criteria?
- What are its advantages and disadvantages?

There are three kinds of bugs to which the author particularly refers:

1. Incorrect statement, concept, code syntax error, or other blunder.
2. Misspelling, misconstruction, or inconsistent usage.
3. Minor typo not likely to cause confusion or misunderstanding.

When reported to Eiffel, the author exemplifies the first two types of bugs from the list above. The following example is given:

- For class 1 and 2 of bugs: the postcondition for a stack.pop
  \texttt{ensure} \ ((\text{pop(push(x,old_stack)))=stack) \ cannot \ be \ directly \ asserted \ in \ Eiffel \ (such \ a \ relation \ can \ be \ wrapped \ in \ a \ Boolean \ function \ however)
4 Classes of Bugs in Eiffel Code

The main purpose of this diploma thesis is to diagnose the bugs found in Eiffel applications. Based on the results of the tests, we have build a bug classification scheme.

4.1 The testing process

The first step in constructing a bug hierarchy is to conduct tests on Eiffel code. The tool that has been used is AutoTest.
Classes in the following libraries and applications have been tested:

- Eiffel Base - basic library of Eiffel
- Gobo Eiffel - application that provides the Eiffel community with free and portable Eiffel tools and libraries. I have worked with the mathematical and xml libraries.
- Espec - mathematical library for Perfect Developer
- EWG - Eiffel Wrapper Generator: a tool that generates Eiffel wrapper classes for C libraries.
- DoctorC: a web service that translates C declarations into English.

4.2 The debugging process

Each test result was followed by a long and demanding process of interpreting the code that generated the bug, the fault reported by AutoTest and the situation as a whole.

The scientific method of debugging described in Section Theoretical Background - The scientific method was used. Observing a failure dealt with analyzing the bug that AutoTest has reported. For each found bug, a hypothesis has been created and predictions were made about the situations in which it can appear, about external and internal factors that had to do with the fault. Each test case was run in the Eiffel Studio debugger. A debugging snapshot in Eiffel Studio is given in Figure 4.1.
Figure 4.1: Debugging in Eiffel Studio
4 Classes of Bugs in Eiffel Code

4.3 Types of faults in Eiffel

The testing and debugging processes mentioned above lead to a classification of faults. Each category of bugs will be explained in the sections to follow.

4.3.1 Implementation/Specification

The starting point in the classification scheme was from the exact software development life cycle. Briefly, it is depicted in Figure 4.2.

![Figure 4.2: Analysis and implementation - schematic view](image)

When building a component or a program one starts from the intended specification. By performing an analysis process the exact (real) specification is formed. During analysis the validation step is also applied. Validation tries to answer the question: “Does the program conform to the specification?”.

Implementation is the next step, after obtaining the real specification of the system. During implementation verification is applied for checking the correctness of the product. Verification tries to answer the question: “Does the program do what it is supposed to?”

Figure 4.3 shows where in the software development process the two types of bugs referred in this section, appear.
The faults that can appear in the validation phase have been classified as *specification bugs*, and the ones that appear in the verification phase as *implementation bugs*.

Next definition, explanations and examples will be given for each of the two categories.

**Specification**

A specification bug appears because of the discrepancy between the intended specification and the real specification.

The example considered is the routine `make` of the class `ARRAY` from the cluster `base.kernel`. This routine allocates space for an array. Its implementation is:

```eiffel
make (min_index, max_index: INTEGER) is
   −− Allocate array; set index interval to
   −− ‘min_index’ .. ‘max_index’; set all values to default.
   −− (Make array empty if ‘min_index’ = ‘max_index’ + 1).
require
   valid_bounds: min_index <= max_index + 1
do
   lower := min_index
   upper := max_index
   if min_index <= max_index then
```

Figure 4.3: Implementation and specification bugs
make_area (max_index − min_index + 1)
else
make_area (0)
end
ensure
  lower_set: lower = min_index
  upper_set: upper = max_index
  items_set: all_default
end

The problem appears in a test case like:

Test case for make in ARRAY
create {ARRAY[ANY]} v_25.make (2147483647)

The specification should also say something about the case in which min_index > max_index + 1, so the routine has a too weak precondition.

**Implementation**

An implementation bug appears because the implementation of the routine does not fulfill the exact specification of the routine and there is no discrepancy between the intended and the real specification.

What is considered to be part of the specification is the precondition, postcondition, class invariant or just what is stated in the comment of a routine. Faults have been generated because:

(a) the specification as it results from the comment of the routine is not respected by the implementation.
(b) the specification as it results from the precondition of the routine is not respected by the implementation.
(c) the specification as it results from the postcondition of the routine is not respected by the implementation.
(d) the specification as it results from the class invariants is not respected by the implementation.
(e) the check instruction is used badly.
(f) there are situations in which a routine \( r_1 \) calls another routine \( r_2 \) with such combination of arguments for \( r_2 \) that it leads to a contract violation in \( r_2 \) (supposing the specification of \( r_2 \) is correct). This situation has been considered as being an implementation problem in \( r_1 \).

Next examples for each of the above mentioned situations will be provided.

The first case to be considered is the one in which the implementation does not respect the precondition of a routine. Take for example the routine `prune_all` of the class `PART_SORTED_SET` from the cluster `base.structures.set`. This routine takes one argument and removes all the items identical to that argument from the object it is called on.

The implementation of `prune_all` is:

```eiffel
prune_all (v: like item) is
   −− Remove all items identical to ‘v’.
   −− (Reference or object equality,
   −− based on ‘object.comparison’.)
   −− Leave cursor ‘off’.
   −− (from PART_SORTED_TWO_WAY_LIST)
require -- from COLLECTION
   prunable: prunable
do
   from
   start
   search (v)
   until
   off or else v < item
loop
   remove
end
if not off then
   finish
end
ensure -- from COLLECTION
   no_more_occurrences: not has (v)
ensure then -- from DYNAMIC_CHAIN
   is_exhausted: exhausted
end
```

The test case that has generated a fault is:

```eiffel
create {PART_SORTED_SET [PART_COMPARABLE]} v1.make
```
The problem that AutoTest reports is:

\begin{verbatim}
PART_SORTED_SET  prune_all @4  Segmentation fault:
<00000000B7A2AD20>  (From PART_SORTED_TWO WAY_LIST)
\end{verbatim}

Operating system signal

The problem appears because there is a call like `Void < Void` in `v < item` in the following piece of code from `prune_all`:

\begin{verbatim}
do
  from
    start
    search (v)
  until
    off or else v < item
loop
  remove
end
\end{verbatim}

This is an implementation problem because considering the specification: Remove all items identical to “v”. Reference or object equality, based on “object_comparison”. Leave cursor “off”. the implementation should ensure that the routine behaves accordingly to its specification even if its argument is `Void`.

The next example presents the situation in which the implementation does not respect the contract stated by the postcondition of a feature.

Consider the routine `abs` of the class `BASIC_ROUTINES` from the cluster `base.kernel.classic`. Its implementation is:

\begin{verbatim}
abs (n: INTEGER): INTEGER is
  -- Absolute value of ‘n’
do
  if n < 0 then
    Result := - n
  else
    Result := n
  end
ensure
  non_negative_result: Result >= 0
end
\end{verbatim}

The test case that generates a fault is:
create \{BASIC\_ROUTINES\} \(v_{15}\)

\(v_{16} := v_{15}.\text{abs}(-2147483648)\)

AutoTest reports the following bug:

BASIC\_ROUTINES \ abs \@1 \\ non\_negative\_result:
\<00000000B7984690> Postcondition violated.

The implementation does not respect the specification (stated by the precondition \textit{non\_negative\_result}). The problem is that, when the routine \texttt{abs} is called with an argument equal to the smallest integer (i.e. \(-2147483648\)) by changing the sign and making \(-(-2147483648)\) the result becomes positive, but it cannot be represented on 32 bits used by \texttt{INTEGER}, hence for an \texttt{INTEGER} representation it is considered negative.

Another type of implementation problem can appear because of the discrepancy between what the class invariant says and what the routine actually does. An outstanding example would be the routine \texttt{make} of the class \texttt{PE\_ARRAY} from the mathematical library of PerfectDeveloper. The class is an array class that simulates the value semantics of Perfect Developer by having client-supplier relationship with ARRAY.

The implementation of the routine \texttt{make} is:

\begin{verbatim}
\texttt{make} (\texttt{min\_index}, \texttt{max\_index}: \texttt{INTEGER}) is
  \begin{itemize}
    \item Allocate array; set index interval to
    \item \texttt{\`min\_index'} .. \texttt{\`max\_index'}; set all values to default.
    \item (Make array empty if \texttt{\`min\_index'} = \texttt{\`max\_index'} + 1).
  \end{itemize}
\texttt{\from ARRAY}
\texttt{valid\_bounds}: \texttt{min\_index} <\= \texttt{max\_index} + 1
\texttt{do}
  \texttt{lower} := \texttt{min\_index}
  \texttt{upper} := \texttt{max\_index}
  \texttt{if} \texttt{\`min\_index'} \texttt{\`max\_index'} \texttt{then}
    \texttt{make\_area} (\texttt{max\_index} - \texttt{min\_index} + 1)
  \texttt{else}
    \texttt{make\_area} (0)
  \texttt{end}
\texttt{\from ARRAY}
\texttt{\lower\_set}: \texttt{lower} = \texttt{min\_index}
\texttt{\upper\_set}: \texttt{upper} = \texttt{max\_index}
\texttt{\items\_set}: \texttt{all\_default}
\texttt{end}
\end{verbatim}

The test case that generates a faulty behavior is the following:
4 Classes of Bugs in Eiffel Code

create {PE_ARRAY[ANY]} v7.make (1, 1)

AutoTest reports the following bug:

ERL_TYPE_IMP_18_PE_ARRAY

creation_procedure_wrapper_make @10

index_lower_bound:

<00000000B7A37F30> Class invariant violated.

The implementation of the routine does not respect the class invariant \textit{index\_lower\_bound}: \textit{lower} = 0.

One should check if the lower index is zero, and if not create an empty array, or return an appropriate message.

Another encountered situation is the one in which the check instruction is not fulfilled. The following example is relevant in underlining this idea. I will refer to the routine \texttt{ref\_comp} of the class \textit{ML\_MAP} in the cluster \textit{ml\_map} from the PerfectDeveloper library.

The implementation of the routine is the following:

\begin{verbatim}
ref_comp: like Current is
  -- returns a new map exactly the same as current one except it
  -- compares keys and values by reference
local
  new_keys: ML\_SEQ[X]
  new_values: ML\_SEQ[Y]
do
  new_keys := keys.subseq (0, keys.count - 1)
  new_values := values.subseq (0, values.count - 1)
  check
    not (new_keys.object\_comparison or new_values.object\_comparison)
end
create Result.make
Result.make_from_parallel\_seqs (new_keys, new_values)
ensure
  resulting\_map\_compares\_ref: not Result.object\_comparison
  current_map\_comparison\_unchanged: keys.object\_comparison = old keys.object\_comparison and
  then values.object\_comparison = old values.object\_comparison
  resulting\_map\_same\_as\_current\_one\_except\_object\_comparison\_flag: # Result = # Current and
  then Result.for\_all (agent key\_value\_in\_map (?, ?, Current.twin))
end
\end{verbatim}

The test case that has generated the fault is the following:
4 Classes of Bugs in Eiffel Code

create \{LINKED_SET [MLPAIR [ANY, ANY]]\} \textit{v}_11.make
create \{MLSEQ [MLPAIR [ANY, ANY]]\} \textit{v}_12.make \textit{from} \textit{set} (\textit{v}_11)
create \{MLMAP [ANY, ANY]\} \textit{v}_13.make \textit{from} \textit{pair} \textit{seq} (\textit{v}_12)
\textit{v}_32 := \textit{v}_13.obj \textit{comp}
\textit{v}_54 := \textit{v}_12.ref \textit{comp}

and AutoTest reported the following problem:

\texttt{ML\_MAP \texttt{ref\_comp} @30}
\texttt{<00000000B799E490> Assertion violated.}

The problem is that the statements in the \texttt{check: not (new\_keys.object\_comparison}
or \texttt{new\_values.object\_comparison)} do not hold at the point where they appear in the routine.

The last example presented in this section is the case of a routine that calls another with a combination of arguments that violate the contracts in the second routine (supposing this one has a correct specification).

Consider \texttt{append} from the class \texttt{BOUNDDED\_STACK} in the cluster \texttt{base.structures.dispenser}. The body of the procedure is:

\texttt{append \texttt{(s: SEQUENCE [G]) is}}
\texttt{−− Append a copy of ‘s’. (Synonym for ‘fill ’) (from DISPENSER)}
\texttt{do}
\texttt{ fill \texttt{(s)}}
\texttt{end}

and the implementation of \texttt{fill} is:

\texttt{fill \texttt{(other: LINEAR [G]) is}}
\texttt{−− Fill with as many items of ‘other’ as possible.}
\texttt{−− Fill items with greatest index from ‘other’ first.}
\texttt{−− Items inserted with lowest index (from ‘other’) will always be on the top of stack.}
\texttt{−− The representations of ‘other’ and current structure need not be the same. (from STACK)}
\texttt{require −− from COLLECTION}
\texttt{ other \texttt{not\_void: other /= Void}}
\texttt{extendible: extendible}
\texttt{local}
\texttt{ temp: ARRAYED\_STACK [G]}
\texttt{do}
\texttt{ create \textit{temp}.make (0)}
\texttt{ from}
\texttt{ other. start}
\texttt{ until}
The test case that produced a contract violation is:

```eiffel
create{ BOUNDED_STACK[ANY]} v_4.make(4)
 v_4.append(Void)
```

and the bug reported by AutoTest is:

```
BOUNDED_STACK fill@4 other_not_void:
<00000000B79B5FF8> (From STACK) Precondition violated.
```

The call `fill (Void)` generates the problem. Fill has a clear specification in the precondition, that its argument should not be `Void`, so the implementation of `append` should take care of the case in which it has a `Void` argument.

**Dunno**

There were situations in which the bug could be interpreted in different ways. We could not tell whether it was an implementation or a specification problem. That is why we have decided to create a new category, which we have called *dunno*.

In order to put a bug in the “dunno” category one has to prove that it can be seen both as an implementation and as a specification bug.

A reasoning procedure in what regards this category will be traced. The query described is `min` from the class `ML_SORT_SEQ` of the cluster `ml_seq` in PerfectDeveloper. The body of `min` is:

```
min: G is
   -- what is the lowest value from the current list ?
   -- if there's more than one such value, return the first occurrence of it
   -- (from ML_SORT_COLLECTION)
```
4 Classes of Bugs in Eiffel Code

require -- from ML_SORT_COLLECTION
  not retrieve_sort_imp.is_empty

local
  current_cell: MLCELL [G]
  current_item: G
  i: INTEGER

do
  from
    i := 0
    current_cell := retrieve_sort_imp.first_cell
    Result := current_cell.item
  until
    i = retrieve_sort_imp.count
loop
  current_item := current_cell.item
  if Result > current_item then
    Result := current_item
  end
  current_cell := current_cell.right
  i := i + 1
end
ensure -- from ML_SORT_COLLECTION
  retrieve_sort_imp.for_all (agent greater_or_equal_y (?, ?, Result))
end

The test case that has generated this fault is the following:

create {ML_SORT_SEQ [COMPARABLE]} v16.make
create {ARRAY [COMPARABLE]} v17.make (-100, -1)
v19 := v16.from_array (v17)
v51 := v19.min

and the bug reported by AutoTest is:

Invalid_object infix ">" @7 other_exists:
<0000000000000000> (From COMPARABLE) Precondition violated.

This fault has two interpretations:

1. implementation problem - because it uses a comparison like Void > Void and if the coder would have read the specification of the infix > operator he would have seen that it cannot be called with Void arguments.

2. specification problem - the specification does not define the behavior of the routine in the case the structure has one or several Void elements.
Considering the above, one could say that the routine \textit{min} belongs to the “dunno” class of faults.

### 4.3.2 Supplier-induced bug

A supplier-induced (or faulty supplier) bug appears in the following cases: suppose routine \textit{r}_1 calls routine \textit{r}_2 (which is faulty). One can have:

- **specification supplier-induced bug**: the problem appears because routine \textit{r}_1 calls routine \textit{r}_2 in one of its contract lines

- **implementation supplier-induced bug**: the problem appears because routine \textit{r}_1 calls \textit{r}_2 in its implementation.

#### Specification supplier-induced bug

Routines that belong to this type can be split into two groups. In the first one there are the ones that call a faulty query or procedure in their precondition, while in the second group there are the ones that call a faulty routine in their postcondition.

A relevant example in this category is the query \texttt{to\_integer} from the class \texttt{STRING} in \texttt{base.kernel}. It calls a faulty routine (\texttt{is\_integer}) in its precondition. Its implementation is:

```eiffel
\texttt{to\_integer : INTEGER is}
\begin{verbatim}
  -- Integer value;
  -- for example, when applied to "123", will yield 123
  require
    \texttt{is\_integer : is\_integer}
  local
    \texttt{Lc : CHARACTER}
    \texttt{l\_is\_negative : BOOLEAN}
    \texttt{Larea : like area}
    \texttt{i, nb : INTEGER}
  do
    \texttt{from}
      \texttt{Larea := area}
      \texttt{nb := count}
      \texttt{i := 0}
    \texttt{until}
      \texttt{Larea.item(i) /= ' '}
  loop
\end{verbatim}
```

40
4 Classes of Bugs in Eiffel Code

\[
i := i + 1
\]
end
\[
L_c := L_{\text{area}.item}(i)
i := i + 1
\]
if \(L_c = '+'\) then
\[
L_c := L_{\text{area}.item}(i)
i := i + 1
\]
elseif \(L_c = '-'\) then
\[
L_{\text{is\_negative}} := \text{True}
L_c := L_{\text{area}.item}(i)
i := i + 1
\]
end
from
\[
\text{Result} := L_c.\text{code} - 48
\]
until
\[
i = nb
\]
loop
\[
L_c := L_{\text{area}.item}(i)
if \(L_c.\text{is\_digit}\) then
\[
\text{Result} := 10 * \text{Result} + L_c.\text{code} - 48
\]
else
\[
i := nb - 1
\]
end
\[
i := i + 1
\]
end
if \(L_{\text{is\_negative}}\) then
\[
\text{Result} := -\text{Result}
\]
end
ensure
\[
\text{single\_digit} : \text{count} = 1 \implies \text{Result} = ("0123456789").\text{index\_of}(\text{item}(1), 1) - 1
\]
\[
\text{minus\_sign\_followed\_by\_single\_digit} : \text{count} = 2 \text{ and } \text{item}(1) = '-' \implies \text{Result} = -\text{substring}(2, 2).\text{to\_integer}
\]
\[
\text{plus\_sign\_followed\_by\_single\_digit} : \text{count} = 2 \text{ and } \text{item}(1) = '+' \implies \text{Result} = \text{substring}(2, 2).\text{to\_integer}
\]
\[
\text{recurse\_to\_reduce\_length} : \text{count} > 2 \text{ or } \text{count} = 2 \text{ and not } ("+-".\text{has}(\text{item}(1))) \implies \text{Result} // 10 = \text{substring}(1, \text{count} - 1).\text{to\_integer} \text{ and}
\[
(\text{Result} \text{\_abs} = \text{substring}(\text{count}, \text{count}).\text{to\_integer})
\]
end

The test case that generated the fault is:

\[
\text{create \{STRING\} v_9.make\_filled('2', 100) v_10 := v_9.\text{to\_integer}}
\]
and the reported bug is:

```
STRING to_integer recurse_to_reduce_length:
<00000000B79F6230> Postcondition violated.
```

The method `to_integer` has as precondition `is_integer` that checks if `Current` represents an `INTEGER`. What it actually does is checking if all the elements in the `STRING` are digits and if it has on the first position a minus or plus sign.

The problem appears in the method `is_integer`. It should also ensure that the `STRING` given as parameter can be represented as an `INTEGER` on 32 bits. The implementation of `is_integer` is:

```eiffel
is_integer : BOOLEAN is
  "-- Does 'Current' represent an INTEGER?
  local
    lc : CHARACTER
    area : like area
    i, nb, l_state : INTEGER
  do
    from
      area := area
      i := 0
      nb := count
    until
      i = nb or l_state = 4
    loop
      lc := area.item (i)
      i := i + 1
      inspect l_state
      when 0 then
        if lc.is_digit then
          l_state := 2
        elseif lc = '-' or lc = '+' then
          l_state := 1
        elseif lc = ' ' then
          else
            l_state := 4
          end
        when 1 then
          if lc.is_digit then
            l_state := 2
          else
```

42
The specification of `is_integer` can be considered as incomplete. If the specification were completed with a statement like “Can Current be represented on 32 bits” a solution for this in the implementation would be to count how many characters the string has and if the number of characters is less than the total number of digits of the maximum integer, then the routine `is_integer` would return `True`.

Considering the above mentioned situation, the routine `to_integer` belongs to the category of specification supplier induced fault.

A specification-induced bug due to a call to a faulty routine in the postcondition occurs in the procedure `extend` of the class `BOUNDED_STACK`. The body of `extend` is:

```
extend (v: like item) is
  -- Push 'v' on top.
require -- from COLLECTION
  extendible: extendible
do
  count := count + 1
  fl . put (v, count)
ensure -- from COLLECTION
  item_inserted: is_inserted (v)
ensure then -- from BAG
  one_more_occurrence: occurrences (v) = old (occurrences (v)) + 1 -- BUG
ensure then -- from STACK
  item_pushed: item = v
```
4 Classes of Bugs in Eiffel Code

end

The test case that revealed the problem is:

```eiffel
create { BOUNDED_STACK[STRING]} bs.make(9)
bs.extend(Void)
```

AutoTest reported the following problem:

```
BOUNDED_STACK  extend @6  one_more_occurrence:
<00000000B7A344D8>  Postcondition violated
```

The problem appears because of the function `occurrences` that counts how many times an item appears in the structure, even if the structure is empty, or has only `Void` elements. So the implementation of `occurrences` should ensure a correct behavior when this query is called on an empty `BOUNDED_STACK`.

**Implementation supplier-induced bug**

Routines belonging to this category call a faulty query or procedure in their lines of implementation (not in their contracts).

An example is `is_inserted` of the class `HEAP_PRIORITY_QUEUE`. Its implementation is:

```eiffel
is_inserted (v: G): BOOLEAN is
  -- Has 'v' been inserted by the most recent insertion?
  -- (By default, the value returned is equivalent to calling 'has (v)'.
  -- However, descendants might be able to provide more efficient implementations.)
do
  Result := has (v)
end
```

The test case that generated the error is:

```eiffel
create { HEAP_PRIORITY_QUEUE [COMPARABLE] } v.2.make (100)
v.9 := v.2. is_inserted (Void)
```

The fault reported by AutoTest is:

```
HEAP_PRIORITY_QUEUE  has @4  not_found_in_empty:
<00000000B7A55A60>  (From ARRAY)  Postcondition violated.
```

As one can notice, `is_inserted` relies on the routine `has` that behaves badly when called on an empty structure with a `Void` argument.
4 Classes of Bugs in Eiffel Code

4.3.3 Inheritance

One of the main issues when dealing with testing is inheritance, one of the essential features of the object-oriented paradigm. But, deep and wide inheritance hierarchies can defy comprehension leading to bugs and reducing testability. As underlined in [3], inheritance can be used to implement specialization relationships or as a programming convenience. Implementation specialization should correspond to problem domain specialization. Reusability of superclass test cases is predicated on this kind of correspondence. In most cases, convenience subclasses will not reflect a true specialization relationship. As such, it is unlikely they can be excused from testing by testing their superclass, even when a lexical excuse can be found. So inheritance must be considered to decide the prudent extent of (re)testing for a subclass. The state model of a root superclass need not consider inherited features because there are none. Subclass behavior, however, is determined by both inherited and local features. If sequential constraints exist on two or more levels, the behavior of the lower levels is a composite of these levels.

Another important issue is overriding under inheritance. In [3], Binder says that the contract metaphor suggests that overriding under inheritance is similar to the business relationship between a prime contractor and a subcontractor. A subcontractor is expected to fulfill some part of the prime contractor’s promises and to refrain from breaking any of the prime contractors’s commitments. The subcontractor is hired to perform a special task that is within the scope of the overall responsibilities of the prime contractor. Similarly, we expect a subclass to accept all valid superclass messages and respond in a way that is consistent with the superclass behavior. The Liskov substitution principle (LSP) gathers a set of conditions needed for achieving consistent subcontracting. These conditions guarantee that:

- any message accepted by a superclass routine will also be accepted by the corresponding subclass method

- the corresponding superclass postcondition and superclass invariant will hold after a message is accepted by a subclass

The concise statement of LSP is that “the objects of the subtype ought to behave the same as those of the supertype as far as anyone or any program using supertype objects can tell”.

Assertions that implement LSP will prevent and detect inheritance bugs. From the contract point of view, a subclass can violate its superclass’s contract in two ways:
1. the subclass implements a stronger precondition, hence a message that would be accepted by the superclass may be rejected by the subclass.

2. if the subclass implements a weaker postcondition or invariant, a state that is invalid for the superclass could be computed by the subclass.

As part of this thesis, we also provide a classification of bugs related to inherited routines. First of all, inherited routines can be modified or not by the subclass. As in Eiffel we have contracts, the issue that came in the discussion was “where does the modification appear”. An inherited routine can modify the original one in the implementation, or in its lines of contract. This reasoning lead to a set of classes of inherited routines. One thing that should be kept in mind when we talk about this classification, is that the inheritance chain can be a long and complex structure.

The types of inherited routines are:

- Stable routines: they are inherited routines which are not changed in any way (no modification in the whole inheritance hierarchy). One of the major questions that appears here is whether these routines should be tested or not. AutoTest defines test cases for all inherited routines, even if they are not modified.

- Not changed routines: they are not modified, but they use modified routines (modified in contract or implementation).

- Mutable routines: they are redefined or effected (their implementation has changed).

- Routines not modified in implementation, but modified in specification (contract).

- Routines modified both in implementation and in their contracts.

In the test cases generated by AutoTest we have found two types of inheritance problems. There are situations in which a routine is inherited from several classes, so the preconditions are or-ed. Consider the example of the routine `append` from the class `LINKED_QUEUE` in the cluster `base.structures.dispenser`. The implementation of the routine is the following:

```plaintext
append (s: SEQUENCE [G]) is
    -- Append a copy of ‘s’. (Synonym for ‘fill ’) (from DISPENSER)
    require -- from SEQUENCE
        argument not void: s /= Void
    do
        fill (s)
```

46
4 Classes of Bugs in Eiffel Code

The test case that has generated the fault is the following:

```eiffel
create {LINKED_QUEUE [ANY]} v_1.make
v_1.remove
v_1.append (Void)
```

AutoTest reported the bug:

```plaintext
LINKED_QUEUE fill @4 other_not_void:
<00000000B7A55A38> (From COLLECTION) Precondition violated.
```

The problem is the fact that the argument is Void and a call like `fill(Void)` generates a precondition violation in `fill`.

If we take a close look at the inheritance tree, we will see that `append` inherits on two branches, from `DISPENSER` (on the branch that inherits from `QUEUE`) and from `SEQUENCE` (on the branch that inherits from `LINKED_LIST`) as depicted in Figure 4.4.

![Inheritance hierarchy for routine append in LINKED_QUEUE.](image)

The version inherited from `SEQUENCE` (on the branch with `LINKED_LIST`) is undefined, so the implementation will be the one from `DISPENSER`. In `DISPENSER`, the implementation of `append` is:
4 Classes of Bugs in Eiffel Code

append (s: SEQUENCE [G]) is
   −− Append a copy of ‘s’.
   −− (Synonym for ‘fill’)
do
   fill (s)
end

The precondition here is True. The precondition from the other branch is also inherited, and this results in the implementation of append from LINKED_QUEUE:

append (s: SEQUENCE [G]) is
   −− Append a copy of ‘s’. (Synonym for ‘fill’ ) (from DISPENSER)
require
   True −− from DISPENSER !!!
   argument not void: s /= Void −− from SEQUENCE
do...
end

As preconditions are or-ed: True or False results in True.

Another type of bug discovered by AutoTest is the one for which the class invariant is modified, and this modification leads to a fault in an inherited routine.

Take the example of default_create from the class ML_SORT_SET from the mathematical library of PerfectDeveloper. The implementation does nothing, as shown by the following listing:

default_create is
   −− Process instances of classes with no creation clause.
   −− (Default: do nothing.) (from ANY) (export status {NONE})
do
end

The test case that has generated the fault is:

create {ML_SORT_SET [COMPARABLE]} v~t0

and the reported bug is:

Invalid_object prefix "#" @1 Segmentation fault:
<0000000000000000> (From ML_SEQ) Operating system signal.

This is an inheritance problem that would not have been discovered if AutoTest would not test routines that are inherited and not modified (in implementation). The default_create routine is inherited from ANY and not modified. The invariant:
4 Classes of Bugs in Eiffel Code

set_size_not_negative :
  # Current >= 0

makes the call # Current >= 0 and Current.implementation is Void so in # there will be a call like # Void.

Wrong export status

For a good understanding of this type of fault, explanations about what is a selective export in Eiffel as shown in [13] will be given.

In order to restrict the set of clients that can call a certain feature \( h \) there is a possibility for a class to have two or more feature clauses. The class will then be of the form:

class S2 feature
  \( f \) ...
  \( g \) ...
feature \( \{ A, B \} \)
  \( h \) ...
  ...
end

Features \( f \) and \( g \) are available to all clients. Feature \( h \) is available only to \( A \) and \( B \) and to their descendants (the classes that inherit directly or indirectly from \( A \) or \( B \)).

As a special case, if you want to hide a feature \( i \) from all its clients, you may declare it as exported to an empty list of clients (in practice it is advisable to use the class NONE).

class S3 feature \{NONE\}
  \( i \) ...
end

In this case the only permitted calls to \( i \) are unqualified calls of the form \( i (...) \) appearing in the text of a routine of \( S3 \) itself, or one of its descendants.

With respect to the export status I have encountered situations in which creation procedures are exported to ANY, but they should be exported to NONE, because they should only be used as creation procedures.

An example considers the routine \texttt{make\_from\_pe\_array\_reverse\_order} of the class \texttt{ML\_SORT\_SEQ} of the mathematical library in PerfectDeveloper. The implementation is given below:

\begin{verbatim}
make_from_pe_array_reverse_order (a: PE\_ARRAY [G]) is
  -- make an immutable sequence of items similar to items of 'a' in order
\end{verbatim}
4 Classes of Bugs in Eiffel Code

--- (from ML_SEQ)

```eiffel
local
  new_seq: like Current
do
  make
  create new_seq.make
  new_seq := new_seq.from_pe_array_reverse_order (a)
  set_first_cell (new_seq.first_cell)
  set_last_cell (new_seq.last_cell)
  secret_count := a.count
end
```

The test case that has generated the failure is:

```eiffel
create {ML_SORT_SEQ [COMPARABLE]} v_1.make
v_1 := v_1.extended.ascending_with (Void)
create {PE_ARRAY [COMPARABLE]} v_10
v_4.make_from_pe_array_reverse_order (v_10)
```

and the reported bug is:

```
ML_SORT_SEQ     make @4     empty:
<00000000B7A42740> (From ML_SEQ)        Postcondition violated
```

The problem is that the routine `make_from_pe_array_reverse_order` should be exported to NONE, so that it could be used only as creation procedure. In this case, the structure is not empty and this will lead to a postcondition violation in the routine `make` that is called by `make_from_pe_array_reverse_order`. This is a special case of implementation problem.

Another example is the case in which a FUNCTION or PROCEDURE is created using the default_create method.

### 4.3.4 Special cases

**External fault**

There were cases in which the problem found by AutoTest came from external routines. For a good understanding of the concepts *external routines* as described in [13] will briefly be presented.

There are situations in which one needs to interface his/her software with non-O-O elements, written in such languages as C, Fortran or Pascal. That is why, Eiffel provides support for external routines. They have most of the trappings of a normal
4 Classes of Bugs in Eiffel Code

routine: name, argument list, result type if it is a function, precondition, postcondition if appropriate. Instead of a do clause it will have an external clause stating the language used for the implementation.

An example of fault belonging to this category, appears in the routine bottom_int_div from the class BASIC_ROUTINES of the cluster base.kernel. Its body is given below.

bottom_int_div \( (n_1, n_2: INTEGER) : INTEGER \) is
\( \text{-- Greatest lower bound of the integer division of } 'n_1' \text{ by } 'n_2' \)
external
\( "C | \%"\) eif_misc.hiso"
alias
\( "\backslash b o i n t d i v " \)
end

The test case that has generated the fault is:

create \{BASIC_ROUTINES\} v_14
v_17 := v_14.bottom_int_div(0,0)

and the bug reported by Autotest is:

------------------------------------------------------------------
BASIC_ROUTINES bottom_int_div Floating point exception:
<00000000B7A76690> Floating point exception.
------------------------------------------------------------------

This is clearly a problem because of a division by zero.

Min/Max INTEGER

This type of bug appears in the situations in which the maximum or minimum integer (refering to the type INTEGER -on 32 bits-) is passed as argument to a routine. On a routine call having as argument Max_int one could have a precondition like a variable \( \leq \text{Max_int} + 1 \) (considered as negative number). A special case of this type of error, that appears in many test cases is the situation in which a routine has as argument the maximum integer. Let’s take for example the routine make of the class STRING:

create \{STRING\} v_{-1}.make (2147483647)

The routine make has as argument the greatest integer. In the body of make we encounter a call like:

make_area \( (n + 1) \)
and in our case \( n = 2147483647 \) so we will have a call like `make_area(2147483648)`. The argument of `make_area` is greater than the largest integer, and AutoTest reports a precondition violation in `make_area`

```
make_area (n: INTEGER) is
  −− Creates a special object for ‘n’ entries.
  require
  ngon_negative_argument: n >= 0
  do
    create area.make (n)
  ensure
    area_allocated: area /= Void and then area.count = n
  end
```

As the integer is too large (\( 2147483648_{dec} = 10000000000000000000000000000000_{bin} \)) the number ”becomes” negative and we have a precondition violation.

**Empty names**

This type of error appears in directory-like structures in which we have empty strings as in the following example taken from the routine `empty` of DIRECTORY.

```
create {STRING} v_1.make_empty
create {DIRECTORY} v_2.make (v_1)
...
create {STRING} v_10.make_empty
create {DIRECTORY} v_11.make (v_10)
v_12 := v_11.empty
```

and the reported bug was:

```
DIRECTORY is_empty @4 directory_exists:
<00000000B79D2748> Precondition violated.
```

We have a precondition violation of the routine `is_empty` as a directory with an ’empty’ name already exists. I think the problem is in the routine `make`:

```
make (dn: STRING) is
  −− Create directory object for the directory
  −− of name ‘dn’.
  require
    string_exists : dn /= Void
  do
    name := dn
```
It only checks if the STRING given as argument is not Void, but it should also ensure that the directory to be created does not exist. A solution would be to insert a postcondition clause that deals with this.

Another issue would be to ensure that directories with 'empty' names are not created (a statement for this could be included in the precondition).

**Void call**

Another type of error appears in the case in which we have a Void argument for a routine.

For example in MEMORY one of the buggy routines is `objects_instance_of`. The problem is encountered in:

```eiffel
create {MEMORY} v\_1
v\_9 := v\_1. objects_instance_of (Void)
```

and reported as:

```plaintext
MEMORY dynamic_type Segmentation fault:
<00000000B7A11D80> (From ISE_RUNTIME) Operating system signal.
```

**Void target**

There are situations in which there is a call like Void.routine as in the following example of the routine `fill` in the class `ARRAYED_TREE`.

```eiffel
fill (other: TREE [G]) is
  -- Fill with as many
  -- items of 'other'
  -- as possible. The
  -- representations
  -- of 'other' and
  -- current node
  -- need not be the
  -- same. (from TREE)
  do
    replace (other.item)
    fill_subtree (other)
  end
```
4 Classes of Bugs in Eiffel Code

The test case that has generated this error is:

\[
\text{create } \{ \text{ARRAYED TREE[STRING]} \} \ v_1.\text{make } (0, \text{Void}) \\
\ v_1.\text{fill } (\text{Void})
\]

**Mixed bugs**

There is another type of errors that were encountered, we could say these are mixed bugs. Consider the example of the routine \textit{grow} in the class \textit{STRING}. Its implementation is:

\[
\text{grow (newsize: INTEGER) is} \\
\quad \text{-- Ensure that the capacity is at least ‘newsize’.} \\
\quad \text{do} \\
\quad \quad \text{if newsize} > \text{capacity then} \\
\quad \quad \quad \text{resize (newsize)} \\
\quad \quad \text{end} \\
\quad \text{ensure -- from RESIZABLE} \\
\quad \quad \text{new_capacity. capacity} >= i \\
\quad \text{end}
\]

The test case that has generated the error is:

\[
\text{create } \{ \text{STRING} \} \ v_1.\text{make\_filled (‘%%’, 10)} \\
\ v_1.\text{grow (2147483647)}
\]

and the reported bug is:

\[
\text{SPECIAL aliased\_resized\_area @1} \\
\quad \text{valid\_new\_count:} \\
\quad \langle00000000B7A11278\rangle \quad \text{Precondition violated.}
\]

The implementation of the routine \textit{resize} is:

\[
\text{resize (newsize: INTEGER) is} \\
\quad \text{-- Rearrange string so that it can accommodate} \\
\quad \text{-- at least ‘newsize’ characters.} \\
\quad \text{-- Do not lose any previously entered character.} \\
\quad \text{require} \\
\quad \quad \text{new\_size\_non\_negative: newsize} >= 0 \\
\quad \text{local} \\
\quad \quad \text{area\_count: INTEGER} \\
\quad \text{do} \\
\quad \quad \text{area\_count := area\_count} \\
\quad \quad \text{if newsize} >= \text{area\_count then}
\]

54
if area = empty_area then
    make_area (newsize + 1)
else
    area := area. aliased_resized_area (newsize + 1)
end
end

I would consider this as:

• faulty implementation supplier in the routine grow

• specification problem in resize (it should ensure the fact that it does not call
  make_area with an argument grater than count - because this is the reason why
  there is precondition violation in make_area).

For this detailed description of fault categories found in Eiffel to be complete, one
should mention that the actual classification was the result of a continuous refinement
process. Defining categories (with their appropriate names) was largely discussed with
Ilinca Ciupa and Andreas Leitner.
5 AutoTest

AutoTest is a fully automatic testing tool for Eiffel systems. It implements the concept of push-button testing: one must only provide it an ace/xace file for the system under test and it will then completely automatically generate and run tests on the classes in the system and at the end give information about test results for each feature.

AutoTest relies on the contracts in the Eiffel code: it interprets contract violations as bugs. Since AutoTest cannot check any functionality that is not specified in the contracts, the quality of the test results is highly dependent on the quality (correctness and completeness) of the contracts. [2].

5.1 Installation

For running AutoTest one needs the following libraries:

- ISE Eiffel 5.6
- Erl-G 1.0.0
- Gobo - use the development version available via CVS from SourceForge
- ePosix 2.3 Beta
- Any platform supported by the above

5.2 Command line arguments

AutoTest accepts the following command line arguments:

- –help : Displays the possible command-line options
- –version : Displays currently running version of AutoTest
- –verbose: Displays information about testing progress
• –just-test: It allows you to re-use a previously generated interpreter. If your system under test and the reflectable classes have not changed, you can set this option and it will save you some (significant) time when running AutoTest.

• –disable-manual: Disable manual testing strategy. Default is to enable it.

• –disable-auto: Disable automated testing strategy. Default is to enable it.

• –disable-minimize: Disable automatic minimization of bug reproducing examples. Default is to enable it. AutoTest tries to minimize any bug reproducing examples it found during testing. It does so extracting a so called backward slice from the original test case and then running the extracted slice again through the interpreter.

• –define: For xace variables (see Gobo documentation)

• –output-dir: Everything that AutoTest generates (the interpreter, log file, test results) is created in this directory

• –time-out: How long AutoTest runs in minutes (30 minutes by default)

• ace_filename: Ace/xace file containing the system under test

• class_name+: List of classes (separated by spaces) that will be tested.

5.3 Ace files

As it was mentioned .ace files in the parameter options for AutoTest, I will make here a short presentation for them. Ace files are intensively used by Eiffel systems. An .ace file is mainly a configuration file (a sort of makefile) that drives the whole compilation process. Ace files are automatically generated by EiffelStudio and are platform dependent. A small example of ace file for Linux is presented in Figure 5.1

Other types of files used are .xace files, that have the same role as .ace files but are platform independent. Recently, there have been introduced .ecf (EiffelConfigurationFile) files that contain the definition of the system to compile, with eventually several targets (but no information relative to an actual compilation is stored in there).
Figure 5.1: An example of ace file
5.4 AutoTest - behind the scenes

The general idea that AutoTest uses for testing a routine is a succession of the following steps:

1. 1. Generate the target and the argument objects of the call. AutoTest uses a random strategy to generate input data for routine calls performed during testing. This strategy treats basic (primitive) and non-basic types differently:
   - For basic types, values are selected randomly from a fixed set. For instance, for integers, this value set is -1, 0, 1, -2, 2, 3, 4, 5, 6, 7, 8, 9, 10, -10, 100, -100, INTEGER.Min_value, INTEGER.Max_value. Whenever we need an integer, we will randomly select one of these values.
   - For non-basic types, we have to build objects of the corresponding type. This is done by randomly calling one of the creation procedures of the class.

Objects used in tests are accumulated during the testing process: while running the tests, a pool of objects is kept. This pool is enriched and diversified as testing proceeds. Whenever there is a call to a certain routine under test, either the necessary objects (target and arguments) are newly created or the existing ones from the pool are reused. The choice between these 2 options is regulated by a heuristics. Objects used for running a routine are returned to the pool (in the new state) after execution of the test case is finished. Furthermore, in order to diversify the pool, modifiers are regularly called (routines which don’t return a value) on random objects in the pool.

2. If the precondition of the routine is satisfied, its body is executed.

3. If a contract violation occurs (in the body of the tested routine or further down the call chain), a bug has been found. If no such violation occurs, the test case has passed.

For a better understanding consider the example:

```java
class STRING
...
valid_index (i: INTEGER): BOOLEAN is
   -- Is ‘i’ within the bounds of the string?
require
   index_positive: i >=0
```
5 AutoTest

```plaintext
do
    Result := (i > 0) and (i <= count)
ensure then -- from INDEXABLE
    only if in index set : Result implies ((i >= index_set.lower)
        and (i <= index_set.upper))
end
... end
```

In order to call the routine, a target object should be created first (an instance of the enclosing class of the routine, STRING) and an argument (an integer number). If the input generator provides a negative value (for example -10), the precondition is violated, so the routine cannot be run. This is called an invalid test case. Suppose the input data generator gives a positive value (for example 20) for the argument. This satisfies the precondition, so the body can be executed. Supposing that the routine valid_index calls other routines and if any contract is violated in those routines, a bug has been found. If the routine does not fulfill the postcondition or it breaks the class invariant, again a fault has been found.

All the above mentioned steps are performed by AutoTest. A user must only specify the class to test (or the routine) and the testing will be done in the manner described.

When the testing is finished, the test results are classified as:

- **pass**: a source code construct is classified as pass when all valid test cases passed.
  
  A test case is classified as pass when it ran successfully.

- **fail**: a source code construct is classified as fail if at least one of its test cases failed.
  
  A test case is classified as fail when during its execution a bug in the system under test was revealed.

- **invalid**: a source code construct is classified as invalid when all its test cases were invalid. A test case is classified as invalid when it could not be run due to a missing prerequisite such as a satisfied precondition.

- **bad response**: a source code construct is classified as bad response when all test cases run for it were either invalid or a bad response. A test case is classified as bad response when during its execution the interpreter on which the test case is executed was not able to respond or responded in an unexpected way.

- **untested**: a source code construct is classified as untested when none of its test cases could be run.
5.5 Test execution: the driver and the interpreter

As described in [2], AutoTest uses a 2-process model when executing tests:

- The process called "interpreter" recognizes and executes very simple Eiffel-like commands and returns a status message to reflect the outcome of the execution. See the next section for more information on the interpreter.

- The driver is the process that contains the testing logic. It communicates with the interpreter via standard I/O and passes to it the simple Eiffel-like commands that the latter understands. In other words, the driver knows the testing strategy and the interpreter is responsible for the actual execution of the tests.

This separation allows for increased robustness of the testing process. If executing one of the routines under test produces an error from which the process performing the call can’t recover, in our model only the interpreter will be affected. The driver will wait for a message from the interpreter for a fixed time, and then kill the interpreter process and restart it, resuming testing from where it left off. If several calls to the same routine cause the interpreter process to become unstable, the driver can decide to abandon testing that routine altogether and go on to testing the next one.

5.5.1 The interpreter

The interpreter is a component that can be used on its own and is also delivered with the Eiffel Reflection Library Generator (ERL-G). It can execute simple commands and returns status messages. For more information on these commands and messages see the documentation available with the ERL-G tool.

The interpreter uses ERL-G to be able to dynamically call routines. The system under test is given as input to ERL-G, which generates corresponding reflectable types for all alive types in the system. Types are considered to be alive according to the following algorithm:

1. Types corresponding to the classes under test (given on the command-line) are alive. When the user wants to test a generic class, the generic parameter(s) will be instantiated with objects of the constraining type (or one of its closest effective descendants if the constraining type is deferred).

2. Types in the transitive closure of the above-mentioned types are alive.
The interpreter then uses these generated reflectable types to call the features under test.

Both the driver and the interpreter write the messages that they exchange during test execution to log files. The logs can be found in directory auto_test_gen/log under the names proxy_log.txt and interpreter_log.txt.

The proxy is the part of the driver process whose only job is to interact with the interpreter. In the proxy log you can see the commands that the driver issued to the interpreter, and the responses that it got back. The latter are listed as comments.

In the interpreter log you can see a list of the requests that the interpreter received, and information about any exceptions thrown during execution of these requests.

5.6 AutoTest Output

The output generated by AutoTest in .html format. It contains informations about the state of the tested routine, total number of tests run, amount of failed, passed, invalid or bad response cases.

A small output example is given in Figure 5.2.

![AutoTest statistics for system 'sample'](image)

Figure 5.2: AutoTest html results
6 Implementation

The implementation was carried out using Eiffel. The main paradigms and concepts of Eiffel were introduced in the chapter *Theoretical Background*.

The applications have been developed under Debian GNU-Linux.

6.1 Testing a routine of a class

A small contribution to AutoTest was to enlarge the set of command line options. The possibility of testing a single feature of a class has been added. In order to do this, the user must introduce the class name followed by a `.` and the routine name. An optional priority argument can be given (it says what priority does the routine have).

For example, in order to test the routine `index_of` from the class `STRING` one should run the following command:

```
auto_test --verbose --time-out=30 string_test.ace STRING.index_of=20
```

6.2 System Design

In what follows it will be shown how does testing the routine of a class has been integrated in the whole system design of AutoTest.

For this, an explanation about the internal architecture of AutoTest is provided. The root class of the system is `AUTO_TEST` and the creation procedure is `execute`. For the reader to understand the way AutoTest works, a trace of this procedure will be done. The main steps are:

- Reset the next option position, that is used for processing the arguments of the program and set the program name and the output directory:

  ```
  make
  Arguments.set_program_name ("auto_test")
  output_dirname := "auto_test_gen"
  ```

```
6 Implementation

- take care of timing settings, and set the default time-out to 30 minutes:

  ```make
  create time_out.make (0, 0, 0, 30, 0)
  start_time := system_clock.date_time_now
  ```

- create an error handler: `create error_handler.make`

- process the input arguments and set the class names and routines to be tested (this will be further detailed).

- compile the ace file and generate the universe (detailed below)

- generate interpreter

- update remaining time

- execute tests

- build result repository

- if minimization is enabled, perform minimization and build the result repository

- generate statistics (in .html format)

The whole implementation of the routine is given in the Appendix. A class diagram of the system is provided in Figure 6.1.

6.2.1 Process the input arguments

The procedure that takes care of processing the input arguments is `process_arguments`. It reads the input parameters one by one and sets the tasks that AutoTest has. It compares the read argument with the set of options ("help", "version", "verbose", "just-test", "disable-manual" etc) and sets the corresponding boolean values of the class `AUTO_TEST_COMMAND_LINE_PARSER`. This routine also initializes the structures corresponding to the class names to be tested.

6.2.2 Compilation

The next step is to compile the system using the `.ace/.xace` configuration files given as argument to AutoTest.

As presented in [8], the steps of the compilation are:
6 Implementation

Figure 6.1: AutoTest Classes

- Degree 6: This is the place where using the configuration file, the file system is traversed for looking at files with the .e extension, and then lookup the class name associated with the .e file.

- Degree 5: This is where the compilation actually starts. Starting from a set of classes that needs to be compiled (e.g. ANY, STRING, INTEGER, ...) and the root class, each class is parsed, and each unparsed class encountered in the parsed class will be added to the set of classes that needs to be compiled. Degree 5 will complete when the transitive closure of referenced classes is done, in other word when the set is empty.

Degree 5 parses each class and creates a CLASS_AS node (that is to say an abstract syntax tree representing the class). The inheritance clauses are also initialized, so that a topological sort can be done at the very end of degree 5.

The topological sort will sort so that when you have a class, all the parents of that class are located before that class in the topological sort. Thus making ANY the first entry (since ancestor of all classes), and NONE the last one (except that since NONE is not a real class, it does not actually appear).

- Degree 4: Using the topological order, the inheritance clause of every classe is analyzed to ensure they are valid, and build their feature table. The feature table
is a table where all available features for a class are registered. It is indexed by name or by routine ID.

- Degree 3: This is where the code for each routine is actually validated, and when it is valid, a BYTE_NODE object representing a compiled version of the AST which is made for code generation purposes is generated. This new object has more information about how to generate certain Eiffel constructs.

6.2.3 Generation of the interpreter

Based on the universe generated at the previous step, an interpreter is created. Mainly, this step consists in generating the types under test and the classes under test for the system.

6.2.4 Execute tests

If manual testing is enabled, manual tests are effected using the procedure execute_manual_tests.

Automatic testing is done using the routine execute_random_tests. It will be largely explained in the implementation section.

6.2.5 The result repository

From the interpreter’s proxy log file created during the execution of the system, a result_repository (instance of AUT_TEST_CASE_RESULT_REPOSITORY) is build. It stores the test case results. If minimization is enabled, these results are minimized at a next step.

6.2.6 Statistics

Using the result_repository an html_generator (instance of AUT_HTML_STATISTICS_GENERATOR) is created. For each class in the pool it constructs a html frame that contains test statistics like:

- total number of tests, how many passed, failed, were invalid or gave a bad response
- the status of each routine in the class
6 Implementation

6.2.7 Particular routine testing

Classes for storing routine information

In order to provide a list of classes with their routines to be tested, we have introduced two classes: `CLASS_NAME_TO_TEST` and `ROUTINE_INFO`.

The class diagram is depicted in Figure 6.2.

![Class diagram for the auto_test cluster](image)

**ROUTINE_INFO**

This class stores information about a routine:

- name of the routine
- priority of testing

It has two creation procedures:

- `make_with_name_and_priority (name: STRING; p: INTEGER)` — initializes the name and the priority with the given arguments
6 Implementation

- *make_with_name (name: STRING)* – initializes the name of the routine with the given argument and the priority is set to a default value (10).

**CLASS_NAME_TO_TEST**

The attributes of this class are:

- **class_name**: a STRING containing the name of the class to be tested
- **routine_name**: a linked list of ROUTINE_INFO objects

The creation procedure sets the name of the class to the given argument and builds an empty list of ROUTINE_INFO objects.

**Processing the input arguments**

The routine *process_arguments* as been slightly modified for accepting as last argument a list containing at least one element of the following:

- **CLASS_NAME** to be tested
- **CLASS_NAME.routine_name** – the name of the routine in the class that we want to test
- **CLASS_NAME.routine_name=priority** – the name of the routine to be tested and its priority.

The procedure used for parsing an input argument in the above format is *process crt_arg_for_routine* (arg_name: STRING).

It parses the argument and splits it into tokens based on the '.' and '=' delimiters. If the class token already exists in the list of classes to be tested, the routine is added to the list of ROUTINE_INFO objects of the corresponding class and the priority is set accordingly to the argument. If the class does not appear in the list of existing classes, a new object of type **CLASS_NAME_TO_TEST** is created and added to this list.

**Executing tests**

Another modification was in the procedure *execute_random_tests*.

The types under test are traversed one by one. If one of the types is in the list of partial tested types (i.e. only a subset of the routines of the class corresponding to the
type has to be tested) then in the corresponding queue of the type only those routines are inserted.

The procedure `execute_random_tests` is given in the listing bellow.

```plaintext
execute_random_tests is
   -- Execute random tests.
require
   universe_not_empty: universe /= Void
   interpreter_not_void: interpreter /= Void
   types_under_test_not_void: types_under_test /= Void
   no_type_under_test_void: not types_under_test.has(Void)
local
   cs: DS_LINEAR_CURSOR [ET_BASE_TYPE]
   strategy: AUT_RANDOM_STRATEGY
   feature_: AUT_FEATURE_OF_TYPE
   type_name: STRING
   aux_class_name: CLASS_NAME_TO_TEST
   rs: DS_LINEAR_CURSOR [ROUTINE_INFO]
do
   create strategy.make (universe, interpreter, error_handler)
from
   cs := types_under_test.new_cursor
   cs. start
until
   cs. off
loop
   create(STRING)type_name.make_from_string (cs.item.name.name)
   aux_class_name := search_test_class_by_name(type_name)
if
   aux_class_name.routines_name.is_empty then
      strategy.queue. set_static_priority_of_type (cs.item, 10)
else
   from
   rs := aux_class_name.routines_name.new_cursor
   rs. start
until
   rs. off
loop
   feature_ := create_feature_from_name(cs.item, rs.item.routine_name)
   strategy.queue. set_static_priority_of_feature (feature_, rs.item.priority)
   rs. forth
end
end
cs. forth
end
```

69
6 Implementation

execute_task_until_time_out (strategy)
end

6.3 Bug report tool

For easing the bug report creation a small tool that allows the user create bug reports has been implemented.

6.3.1 Technology

For implementing this tool the EiffelVision2 library was used. It offers an object-oriented framework for graphical user interface (GUI) development. It provides an effective way of building advanced graphical applications using user interface standards and toolkits (such as Microsoft Windows and GTK).

The EiffelVision 2 library includes the following interface clusters:

- A Kernel cluster that includes classes that are key to a Vision2 application. The main class is \textit{EV\_APPLICATION} which is the main entry point of all Vision2 applications.

- A Widgets cluster containing classes that are used to create Vision2 widgets. Widgets are the objects that the user sees and interacts with on the desktop; examples of widgets are windows, buttons and labels.

- An Items cluster includes the classes needed to create items; items can be thought of as widgets that are contained within only a certain type of widget, such as an \textit{EV\_LIST} that may only contain objects of type \textit{EV\_LIST\_ITEM}. Items provide an abstract way of dealing with an item widget’s internal data structures and provide in many cases the same functionality that a widget does.

- An Events cluster containing classes that allow for user-initiated events, such as the clicking of a button to be dealt with via the use of a linked list of agents (\textit{EV\_ACTION\_SEQUENCE}). Agents can be thought of as an object that encapsulates a certain routine. When a user clicks a button on the screen, the corresponding \textit{EV\_BUTTON} object has its associated \textit{pointer\_button\_press\_actions} fired and this in turn, fires all of the agents held within, thus calling all of the procedures represented by the agents. Every widget and item has a certain number of \textit{ACTION\_SEQUENCE} objects that are linked with a certain type of event.
• A Properties cluster contains classes that allow for the customization of Vision 2 widgets and items. Classes such as \texttt{EV \_COLORIZABLE} and \texttt{EV \_FONTABLE} contain routines that allow for color and fonts to be altered for a widget or item respectively.

• A Support cluster includes classes that provide more professional touches to an application, whether these are keyboard shortcuts (\texttt{EV \_ACCELERATOR \_LIST}) or graphical output (\texttt{EV \_GRAPHICAL \_FORMAT}) for drawable widgets such as \texttt{EV \_PIXMAP}.

• A Figures cluster that allows for the projection of two-dimensional shapes (figures) on to an \texttt{EV \_DRAWABLE} or printer via the use of an \texttt{EV \_PROJECTOR}.

6.3.2 Design

The development of the BugReport tool is based on a multiple layer development. The layers involved are the following:

• graphical user interface (GUI)

• process/data management

• storage

A schematic view of the classes and clusters involved in the design is given in Figure 6.3.

The root class of the system is \texttt{BUG \_REPORT \_GUI}. It contains an instance of the class \texttt{MAIN \_WINDOW}.

The creation procedure of \texttt{BUG \_REPORT \_GUI} is the routine \texttt{make \_and \_launch}. It creates and shows the instance of \texttt{MAIN \_WINDOW} and starts the application by calling the routine \texttt{launch}. This begins the event processing loop.

creation procedure for \texttt{BUG \_REPORT \_GUI}

\begin{verbatim}
create
make_and_launch
feature \{NONE\} -- Initialization
make_and_launch is
    -- Create 'Current', build and display 'main_window',
    -- then launch the application.
do
\end{verbatim}
Figure 6.3: Bug Report Tool - class diagram
That class that deals with the actual graphical interface, is `MAIN_WINDOW`. It implements user action functions, like creating a new cluster report, introducing details for the routine of a selected class, modifying existing description features.

The class diagram is presented in figure 6.4.

![Class Diagram](image)

**Figure 6.4: Bug Report Tool - class diagram**

### 6.3.3 User Interface Functionality

I have implemented the following features for creating a bug report:

1. create a cluster detailed report
2. create a class report
3. create a routine report
Figure 6.5: Interface of the bug report tool
Creating a cluster report

The interface for creating a new cluster detailed report is given in the Figure 6.6. In order to create a new cluster report, the user has to insert the name of the cluster for which the description is done. Then, for this cluster classes can be added in a tree list area. By selecting a class in this list, the user has the possibility of adding routines to it. For a routine, the following data is stored:

- routine name
- type of bug encountered in the routine
- remarks - if for example the bug is a supplier induced one then the user can specify on which faulty routine it relies on
- description - a text field that offers the possibility of inserting an explanation for the reason why the bug was considered as belonging to a certain category
- test case - the piece of code that has generated the fault
- reported bug - the output of AutoTest with respect to the encountered fault.

Additional dialogs

Several constraints are imposed when introducing data. For example, the name of the cluster, class or routine cannot be void, so whenever this happens a pop-up dialog is shown, for the user to know what he/she has done wrong, and why the intended operation did not succeed. This will help users recognize, diagnose and recover from error. A small example is depicted in the Figure

Figure 6.6: Additional error/warning-message dialogs
Visibility of system status

A status field informs the users at each moment about the operation that is being performed, so they will be informed about what is going on in the system.

6.3.4 The process management layer

At this level we have the classes:

- CLASS_DATA
- ROUTINE_DATA

The class ROUTINE_DATA contains information about a routine:

- name
- category of faults it belongs to the bug)
- test case
- description
- an EV_TREE_ITEM object that represents the tree subitem in the tree of classes in the graphical user interface.

The class CLASS_DATA contains information about the class and its routines:

- class name
- the EV_TREE_ITEM that is referred in the tree item GUI
- a list of routines that appear in the class and for which the bug description is done.

The operations that can be performed on it are:

- creation
- addition of a routine to a class
- deletion of a routine
- modification of a routine
6.3.5 The storage layer

The storage layer is represented by a file storage operation. For the user to have the information collected and displayed in a friendly and easy to understand manner, we have added the functionality for generating .pdf (and implicitly .tex files).

In order to generate a file report the user just needs to click on the Generate Latex button. The whole process that happens behind the scenes is the following:

- create a file (with the path chosen by the user - a file chooser gui element is used for this)
- traverse the list of classes, and for each class traverse the list of routines. Put the information for each routine in the .tex file, respecting the format and commands that \LaTeX has.
- after the .tex file is closed, the user needs to run a Makefile script that will convert the .tex file into .dvi, then .ps and .pdf.
7 Results

This chapter provides a schematic view of the results we had when running AutoTest and applying to each bug the classification scheme.

7.1 EiffelBase

The EiffelBase library is the one that has been widely tested and important results have been obtained with it. That is why a list with numbers of bugs is provided bellow.

- Implementation bugs: 27
- Specification bugs: 28
- Dunno bugs: 2
- Implementation supplier-induced bugs: 30
- Specification supplier-induced bugs: 9
- Inheritance: 12 (1 implementation, 11 specification)
- Wrong export status: 19 (all specification)
- External fault: 6 (2 specification, 4 implementation)
- Void target: 5 (3 specification, 2 implementation)

A detailed result, with classes and routines is given in the Appendix. A graphical result of the distribution of bugs in EiffelBase is given in Figure 7.1.

As one can notice, a large number of bugs were supplier-induced. An important thing to mention here is that all these faults would disappear if the bug in the routine they rely on is corrected.
7.2 General results

Next a result of the total number of bugs and tests found in the applications and libraries that have been tested is given:

<table>
<thead>
<tr>
<th>Library/ Application</th>
<th>Library/ Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>EiffelBase base.kernel base.support base.structures</td>
<td>Gobo Gobo -xml Gobo - math</td>
</tr>
<tr>
<td>Perfect Devel - math</td>
<td>DoctorC EWG</td>
</tr>
</tbody>
</table>

The mathematical library of PerfectDeveloper provided some interesting results. Several routines were
8 Directions for future work

As possibilities for further development of this project I would mention the following ideas:

- Continue the field case study, and come up with new categories of faults, new interpretations of the existing ones.

- Develop a graphical user interface for AutoTest that will allow a fast and easy usage. Offer the user the option to provide input arguments in a random order.

- Develop a database storage mechanism for the Bug Report Tool. The information stored could include:
  - types of bugs and a short description for each
  - cluster reports, with number of bugs/class
  - class reports, with number of buggy routines
  - routine descriptions with fault category to which it belongs to, test case, AutoTest output
9 Conclusions

The goal of this project was to build a classification scheme of the bugs that can appear in Eiffel code.

The classification scheme that was developed covers several types of bugs. Each bug that belongs to the category supplier-induced, wrong export status, external fault, void target, inheritance, can be considered as being a particular case of implementation or specification bug.

An important achievement is related to inherited routines. By finding bugs in inherited routines that are not modified in implementation, we have proved that testing inherited and not modified routines is a step that should be done. (In the literature this is called \textit{The fragile base class problem}). Several testing tools do not take into consideration stable routines when testing, even this is necessary (as the results show).

Most of the inheritance bugs dealt with a faulty inherited specification. From this point of view when writing code, one should analyze the contracts in the inheritance tree, and ensure the right preconditions/postconditions are inherited in a routine.

All the bugs in the supplier-induced (or faulty supplier) category will disappear if the fault is corrected in the routine they rely on.

Running AutoTest on a large number of classes revealed some important issues about it.

- There were bugs that AutoTest could not discover. For example, while debugging a routine that was classified as buggy by AutoTest, we have realized that the problem is not in that particular routine, but in one on which it relies (that was declared OK in the tests run by AutoTest).

- A large percentage of the bugs were reported precisely and this eased the debugging and interpretation work.

- A weak point of this tool is dealing with classes that have many dependents. It takes about 3h to perform a test and the compilation process consumes many
9 Conclusions

CPU resources. Also, there were situations in which the interpreter and the driver blocked.
Bibliography


10 Appendix

10.1 AutoTest Implementations

10.1.1 Creation procedure

```erlang
execute is
-- Start 'erl_gen' execution.
local
  a_file : KL_TEXT_INPUT_FILE
  a_lace_parser: ET_LACE_PARSER
  an_out_lace_parser: AUT_LACE_PARSER
  a_lace_error_handler: ET_LACE_ERROR_HANDLER
  an_xace_parser: ET_XACE_UNIVERSE_PARSER
  an_xace_error_handler: ET_XACE_DEFAULT_ERROR_HANDLER
  an_xace_variables: DS_HASH_TABLE [STRING, STRING]
  gobo_eiffel : STRING
  nb: INTEGER

do
  make
  arguments.set_program_name ("auto_test")
  output_dirname := "auto_test_gen"
  create time_out.make (0, 0, 0, 0, 30, 0)
  start_time := system_clock.date_time_now
  create error_handler.make
  process_arguments
  if class_names.count = 0 then
    error_handler. report_no_classes_specified_error
    exceptions.die (1)
  end
  create a_file.make (ace_filename)
  a_file.open_read
  if a_file.is_open_read then
    nb := ace_filename.count
    if nb > 5 and then ace_filename.substring (nb - 4, nb).is_equal (".xace") then
```

86
create an_race_error_handler.make_standard
create an_race_variables.make_map (100)
an_race_variables. set_key_equality_tester ( string_equality_tester )
gobo_eiffel := execution_environment.variable_value ("GOBO_EIFFEL")
if gobo_eiffel /= Void then
   an_race_variables. force_last ( gobo_eiffel , "GOBO_EIFFEL")
end
an_race_variables. force_last ( "True", "ERL_G_NOIMPLEMENTATION")
if defined_variables /= Void then
   parse_defined_variables ( defined_variables , an_race_variables)
end
create an_race_parser.make_with_variables ( an_race_variables , an_race_error_handler)
an_race_parser. parse_file ( a_file )
a_file . close
if not an_race_error_handler.has_error then
   universe := an_race_parser. last_universe
end
else
create a_lace_error_handler.make_standard
create a_lace_parser.make ( a_lace_error_handler)
a_lace_parser. parse_file ( a_file )
a_file . close
if not a_lace_parser. syntax_error then
   universe := a_lace_parser. last_universe
create an_aut_lace_parser.make_standard
a_file . open_read
an_aut_lace_parser. parse_file ( a_file )
a_file . close
external_text := an_aut_lace_parser. external_text
end
end
if universe /= Void then
   setup_universe
   generate_interpreter
   if interpreter /= Void then
      update_remaining_time
      if time_out.second_count > 0 and error_handler.remaining_time.second_count <= 0 then
         error_handler. report_no_time_for_testing ( time_out )
      else
         execute_tests
         build_result_repository
         if is_minimization_enabled then
            print("Minimization Enabled")
            minimize_universe
         end
      end
   end
else
   execute_tests
   build_result_repository
   if is_minimization_enabled then
      print("Minimization Enabled")
      minimize_universe
   end
end
10 Appendix

```
minimize_witnesses
build_result_repository
end
generate_statistics
end
else
  error_handler. report_interpreter_generation_error
end
else
  exceptions. die (3)
end
else
  error_handler. report Cannot read_error (ace_filename)
exceptions. die (1)
end
end
```

10.2 EiffelBase - detailed results

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>base.dispensers.set</td>
<td>PART_SORTED_SET</td>
<td>duplicate item prune all</td>
<td>Impl. Spec Impl.</td>
</tr>
<tr>
<td>ARRAYED_SET</td>
<td>make</td>
<td></td>
<td>Impl. suppl. ind.</td>
</tr>
<tr>
<td>TWO_WAY_SORTED_SET</td>
<td>duplicate item</td>
<td></td>
<td>Impl. suppl. ind item</td>
</tr>
<tr>
<td>LINKED_SET</td>
<td>there_exists</td>
<td></td>
<td>Wrong exp. status</td>
</tr>
<tr>
<td>BINARY_SEARCH_TREE</td>
<td>duplicate</td>
<td></td>
<td>Impl</td>
</tr>
<tr>
<td>BINARY_SEARCH_TREE_SET</td>
<td>duplicate extend</td>
<td></td>
<td>Impl Impl. suppl ind Void target make in BST</td>
</tr>
</tbody>
</table>

88
<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>put</td>
<td>Impl. suppl ind</td>
<td></td>
<td>make in BST</td>
</tr>
</tbody>
</table>

**base.structures.dispensers**

<table>
<thead>
<tr>
<th></th>
<th>Impl. suppl ind</th>
<th>Impl. suppl ind</th>
<th>Spec. suppl ind</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>isInserted</td>
<td>has</td>
<td>has</td>
<td></td>
<td>has</td>
</tr>
<tr>
<td>make</td>
<td>has</td>
<td></td>
<td>make from ARRAY</td>
<td>has</td>
</tr>
<tr>
<td>extend</td>
<td></td>
<td>Spec suppl ind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>occurrences</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>append</td>
<td>Impl</td>
<td>Spec</td>
<td>Inherit</td>
<td></td>
</tr>
</tbody>
</table>

**LINKED_STACK**

| append              | Impl            |              |                 |          |

**ARRAYED_STACK**

<table>
<thead>
<tr>
<th></th>
<th>Impl suppl ind</th>
<th>check instr</th>
<th>make from ARRAY</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>isInserted</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>make</td>
<td></td>
<td>Impl suppl ind</td>
<td>make from ARRAY</td>
<td></td>
</tr>
<tr>
<td>append</td>
<td>Impl</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ARRAYED_QUEUE**

<table>
<thead>
<tr>
<th></th>
<th>Spec.</th>
<th>Spec suppl ind</th>
<th>index_set</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>has</td>
<td></td>
<td>Spec suppl ind</td>
<td>index_set</td>
<td></td>
</tr>
<tr>
<td>index_set</td>
<td></td>
<td>Spec suppl ind</td>
<td>class invariant</td>
<td></td>
</tr>
<tr>
<td>make</td>
<td></td>
<td>Spec suppl ind</td>
<td>occurrences</td>
<td></td>
</tr>
<tr>
<td>full</td>
<td></td>
<td>Spec suppl ind</td>
<td>occurrences</td>
<td></td>
</tr>
<tr>
<td>extend</td>
<td></td>
<td>Spec Suppl ind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>append</td>
<td>Spec</td>
<td>Inherit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LINKED_QUEUE**

| append              | Spec            | Inherit        |                 |          |

**BOUNDED_QUEUE**

<table>
<thead>
<tr>
<th></th>
<th>Impl suppl ind</th>
<th>Spec suppl ind</th>
<th>make in ARRAY</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td></td>
<td>Spec suppl ind</td>
<td>make in ARRAY</td>
<td></td>
</tr>
<tr>
<td>extend</td>
<td></td>
<td>Spec suppl ind</td>
<td>make in ARRAY</td>
<td></td>
</tr>
<tr>
<td>append</td>
<td>Spec</td>
<td>Inherit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**HEAP_PRIORITY_QUEUE**

<table>
<thead>
<tr>
<th></th>
<th>Spec</th>
<th>Impl</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>index_set</td>
<td></td>
<td>Impl suppl ind</td>
<td>has</td>
<td></td>
</tr>
<tr>
<td>has</td>
<td></td>
<td>Impl suppl ind</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isInserted</td>
<td></td>
<td>Impl suppl ind</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
<td>Spec.</td>
<td>Impl suppl. ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>append</td>
<td>Spec.</td>
<td>Inherit</td>
<td></td>
</tr>
<tr>
<td>prune_all</td>
<td>Spec.</td>
<td>Impl suppl. ind</td>
<td>has</td>
</tr>
<tr>
<td>extend</td>
<td>Spec.</td>
<td>Spec suppl. ind</td>
<td>occurences</td>
</tr>
</tbody>
</table>

#### base.structures.tree

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>Spec.</td>
<td>Impl suppl. ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>prune_all</td>
<td>Spec.</td>
<td>Inherit</td>
<td></td>
</tr>
<tr>
<td>extend</td>
<td>Spec.</td>
<td>Impl suppl. ind</td>
<td>has</td>
</tr>
<tr>
<td></td>
<td>Spec.</td>
<td>Spec suppl. ind</td>
<td>occurences</td>
</tr>
</tbody>
</table>

#### ARRAYED_TREE

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_item</td>
<td>Impl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>valid_cursor_index</td>
<td>Impl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>make</td>
<td>Spec.</td>
<td>Impl suppl. ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>remove_left_child</td>
<td>Spec.</td>
<td>Inherit</td>
<td></td>
</tr>
<tr>
<td>remove_right_child</td>
<td>Spec.</td>
<td>Impl suppl. ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>child_back</td>
<td>Spec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>child_go_to</td>
<td>Spec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>do_all</td>
<td>Spec.</td>
<td>Wrong exp. status</td>
<td></td>
</tr>
<tr>
<td>fill</td>
<td>Spec.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### LINKED_TREE

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid_cursor_index</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fill</td>
<td>Spec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>child_put</td>
<td>Spec.</td>
<td>Inherit</td>
<td></td>
</tr>
</tbody>
</table>

#### TWO_WAY_TREE

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid_cursor_index</td>
<td>Impl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>child_put</td>
<td>Spec.</td>
<td>Inherit</td>
<td></td>
</tr>
<tr>
<td>fill_from_binary</td>
<td>Spec.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### BINARY_TREE

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>valid_cursor_index</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>child_go_to</td>
<td>Spec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>child_remove</td>
<td>Spec.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### FIXED_TREE

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_item</td>
<td>Impl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>index_of</td>
<td>Spec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>valid_cursor_index</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>make</td>
<td>Spec.</td>
<td>Impl suppl ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>make_filled</td>
<td>Spec.</td>
<td>Impl suppl ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>Routine name</td>
<td>Impl. / Spec.</td>
<td>Special case</td>
<td>Remarks</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
<td>--------------</td>
<td>---------</td>
</tr>
<tr>
<td>put_child</td>
<td>Spec</td>
<td>Inherit</td>
<td></td>
</tr>
<tr>
<td>remove_child</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>child_back</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>child_go_to</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>put_nth</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array_make</td>
<td>Impl suppl ind</td>
<td></td>
<td>make from ARRAY</td>
</tr>
</tbody>
</table>

**base.structures.list**

<table>
<thead>
<tr>
<th>ARRAYED_CIRCULAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
</tr>
<tr>
<td>put</td>
</tr>
</tbody>
</table>

**ARRAYED_LIST**

<table>
<thead>
<tr>
<th>is_inserted</th>
<th>Impl</th>
<th>Wrong export status</th>
<th>check instr</th>
</tr>
</thead>
<tbody>
<tr>
<td>there_exists</td>
<td>Spec.</td>
<td>Impl suppl ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>make</td>
<td>Impl</td>
<td>Impl suppl ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>make_filled</td>
<td>Impl</td>
<td>Impl suppl ind</td>
<td>check instr</td>
</tr>
<tr>
<td>put</td>
<td>Spec.</td>
<td>Wrong exp. status</td>
<td></td>
</tr>
</tbody>
</table>

**TWO_WAY_LIST**

| put              | Impl |                     |             |

**SORTED_TWO_WAY_LIST**

| put              | Impl |                     |             |

**TWO_WAY_CIRCULAR**

| put              | Impl |                     |             |

**LINKED_CIRCULAR**

| put              | Impl |                     |             |

**LINKED_LIST**

| put              | Impl |                     |             |

**base.structures.table**

<table>
<thead>
<tr>
<th>HASH_TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>make</td>
</tr>
<tr>
<td>replace_key</td>
</tr>
</tbody>
</table>

**ARRAY2**

| for_all          | Wrong exp. status |             |         |
## 10 Appendix

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>there_exists</td>
<td>Spec.</td>
<td>Wrong exp. status</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>make</td>
<td>Spec.</td>
<td>Impl suppl. ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>force</td>
<td>Spec.</td>
<td>Impl suppl. ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>resize</td>
<td>Spec.</td>
<td>Impl suppl. ind</td>
<td>make from ARRAY</td>
</tr>
<tr>
<td>conservative_resize</td>
<td>Impl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>do_if</td>
<td>Spec.</td>
<td>Wrong exp. status</td>
<td></td>
</tr>
<tr>
<td>do_all</td>
<td>Spec.</td>
<td>Wrong exp. status</td>
<td></td>
</tr>
</tbody>
</table>

### base.support

### C_STRING

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_empty</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### FIBONACCI

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_lower_fibonacci</td>
<td>Spec</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
<tr>
<td>for_all</td>
<td>Spec.</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
<tr>
<td>there_exists</td>
<td>Spec.</td>
<td>spec. suppl. ind. off</td>
<td></td>
</tr>
<tr>
<td>item</td>
<td>Spec.</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
<tr>
<td>do_all</td>
<td>Spec.</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
<tr>
<td>search</td>
<td>Spec.</td>
<td>impl. suppl. ind</td>
<td>item</td>
</tr>
</tbody>
</table>

### PRIMES

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>all_lower_primes</td>
<td>Spec.</td>
<td>impl. suppl. ind</td>
<td>make</td>
</tr>
<tr>
<td>for_all</td>
<td>Spec.</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
<tr>
<td>there_exists</td>
<td>Spec.</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
<tr>
<td>item</td>
<td>Spec.</td>
<td>spec. suppl. ind. off</td>
<td></td>
</tr>
<tr>
<td>do_if</td>
<td>Spec.</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
<tr>
<td>do_all</td>
<td>Spec.</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
<tr>
<td>search</td>
<td>Spec.</td>
<td>impl. supp ind.</td>
<td>search</td>
</tr>
</tbody>
</table>

### RANDOM

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>real_item</td>
<td>Spec</td>
<td>impl. suppl. ind</td>
<td>item</td>
</tr>
<tr>
<td>double_item</td>
<td>Spec</td>
<td>impl. suppl. ind</td>
<td>item</td>
</tr>
<tr>
<td>real_i_th</td>
<td>Spec</td>
<td>impl. suppl. ind</td>
<td>item</td>
</tr>
<tr>
<td>for_all</td>
<td>Spec.</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
<tr>
<td>there_exists</td>
<td>Spec.</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
<tr>
<td>item</td>
<td>Spec.</td>
<td>spec. suppl. ind</td>
<td>off</td>
</tr>
<tr>
<td>do_if</td>
<td>Spec.</td>
<td>Wrong exp.status</td>
<td>callable</td>
</tr>
</tbody>
</table>
### Routine name

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>do_all search off</td>
<td>Spec. / Spec.</td>
<td>Wrong exp. status. impl. suppl. ind item</td>
<td>callable</td>
</tr>
</tbody>
</table>

**base.kernel**

#### STRING

<table>
<thead>
<tr>
<th>Routine</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>adapt to_integer</td>
<td>Spec</td>
<td>Dunno spec. suppl ind</td>
<td>is_integer</td>
</tr>
<tr>
<td>is_integer make</td>
<td>Spec</td>
<td>Impl. suppl ind</td>
<td>make</td>
</tr>
<tr>
<td>make_filled grow resize deep_copy</td>
<td>Spec Impl.</td>
<td>Impl. suppl ind</td>
<td>resize</td>
</tr>
</tbody>
</table>

#### BASIC_ROUTINES

<table>
<thead>
<tr>
<th>Routine</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
</table>

#### EXCEPTIONS

<table>
<thead>
<tr>
<th>Routine</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>raise raise_retreivel_exception</td>
<td>Impl.</td>
<td>External fault</td>
<td></td>
</tr>
<tr>
<td>STORABLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>raise raise_retreivel_exception</td>
<td>Impl.</td>
<td>External fault</td>
<td></td>
</tr>
</tbody>
</table>

#### DIRECTORY

<table>
<thead>
<tr>
<th>Routine</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_open_read</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## 10.3 PerfectDeveloper - Mathematical library

### 10.3.1 Cluster espec-ml_map
10 Appendix

<table>
<thead>
<tr>
<th>Class</th>
<th>#impl LOC</th>
<th>#assert. LOC</th>
<th>#impl bugs</th>
<th>#spec bugs</th>
<th>#fail tests/ total tests</th>
<th>#buggy routines / total</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML_HASH_MAP</td>
<td>46</td>
<td>221</td>
<td>0</td>
<td>9</td>
<td>3.84% (81/2105)</td>
<td>12.5% (9/72)</td>
</tr>
<tr>
<td>ML_MAP</td>
<td>1320</td>
<td>220</td>
<td>2</td>
<td>3</td>
<td>3.96% (80/2018)</td>
<td>7.04% (5/71)</td>
</tr>
<tr>
<td>ML_PAIR</td>
<td>227</td>
<td>45</td>
<td>0</td>
<td>1</td>
<td>0.24% (2/817)</td>
<td>2.63% (1/38)</td>
</tr>
</tbody>
</table>

Routine bug overview

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML_MAP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ref_comp</td>
<td>Impl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_disjoint_from</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extended_by_pair</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intersection</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>make_from_two_lists</td>
<td>Impl.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML_HASH_MAP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>from_hash_table</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>key_value_pair_exists_in_table</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>associates_with_the_same_value</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intersection</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>union</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extended_by_pair</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_disjoint_from</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>make_from_pair_seq</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>make_from_two_lists</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML_PAIR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>from_parallel_lists</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10.3.2 Cluster espec-ml_seq
## Routine bug overview

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ML_SEQ</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>begins_with</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>inserted_before_with_list</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>from_array_reverse_order</td>
<td>Fault. impl. suppl</td>
<td></td>
</tr>
<tr>
<td>from_list_reverse_order</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>from_pe_array</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>from_pe_array_reverse_order</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>from_array_selected_order</td>
<td>Spec</td>
<td>Wrong export status</td>
</tr>
<tr>
<td>from_list_selected_order</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>from_set_selected_order</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>from_pe_array_selected_order</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>from_pe_array_selected_order</td>
<td>Spec</td>
<td>Wrong export status</td>
</tr>
<tr>
<td>hold_count_nested</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>make_from_array_reverse_order</td>
<td>Faulty impl. suppl</td>
<td></td>
</tr>
<tr>
<td>make_from_array_selected_order</td>
<td>Wrong export status</td>
<td></td>
</tr>
<tr>
<td>make_from_list_selected_order</td>
<td>Wrong export status</td>
<td></td>
</tr>
<tr>
<td>make_from_pe_array</td>
<td>Faulty impl. suppl</td>
<td></td>
</tr>
<tr>
<td>make_from_pe_array_reverse_order</td>
<td>Faulty impl. suppl</td>
<td></td>
</tr>
<tr>
<td>make_from_pe_array_selected_order</td>
<td>Faulty impl. suppl</td>
<td></td>
</tr>
<tr>
<td>make_from_set</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>make_from_list</td>
<td>Spec</td>
<td></td>
</tr>
<tr>
<td>make_from_array</td>
<td>Faulty impl. suppl</td>
<td></td>
</tr>
<tr>
<td>from_array</td>
<td>Faulty impl. suppl</td>
<td></td>
</tr>
</tbody>
</table>
### 10.3.3 Cluster espec-ml_set

<table>
<thead>
<tr>
<th>Class</th>
<th>#spec bugs</th>
<th>#impl SI</th>
<th>#inherit bug</th>
<th>#fail tests/ total tests</th>
<th>#buggy routines / total tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML_SET</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>20.06%(62/309)</td>
<td>11.29%(7/62) **</td>
</tr>
<tr>
<td>ML_SORT_SET</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>96.12%(6332/6587)</td>
<td>100%(1/1)</td>
</tr>
</tbody>
</table>
There are two routines that behave badly, but have not been discovered by the test cases generated by AutoTest (see the description of `make_from_array`)

### Routine bug overview

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Impl. / Spec.</th>
<th>Special case</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML_SET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_frame_equal</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_strict_subset_of</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>intersection</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>repeat</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>make_from_set</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>make_from_list</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>make_from_array</td>
<td>Impl. suppl. induced</td>
<td>Impl. suppl. induced</td>
<td></td>
</tr>
<tr>
<td>from_array</td>
<td>Impl. suppl. induced</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extended_by</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML_SORT_SET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>default_create</td>
<td></td>
<td>Inheritance</td>
<td></td>
</tr>
</tbody>
</table>