A SOFTWARE QUALITY WORKBENCH FOR TESTABLE REQUIREMENTS AND SPECIFICATIONS

FARAZ AHMADI TORSHIZI

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF COMPUTER SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE AND ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO
AUGUST 2007

A SOFTWARE QUALITY WORKBENCH FOR TESTABLE REQUIREMENTS AND SPECIFICATIONS

by **Faraz Ahmadi Torshizi**

a thesis submitted to the Faculty of Graduate Studies of York University in partial fulfilment of the requirements for the degree of

MASTER OF COMPUTER SCIENCE

A SOFTWARE QUALITY WORKBENCH FOR TESTABLE REQUIREMENTS AND SPECIFICATIONS

by **Faraz Ahmadi Torshizi**

Examination Committee Members:

1. Dr. Jonathan S. Ostroff
2. Dr. Vassilios Tzerpos
3. Dr. Aijun An
4. Dr. Dong Liang

Abstract

In this thesis, *customer requirements* (in the problem domain) are differentiated from *design specifications* (in the solution space). The design *specification* is the artifact intermediate between the implemented code and the customer *requirements*. We argue that the customer requirements and the design specifications should be testable and testable early in the design cycle leading to early detection of implementation and specification errors. We thus provide a method (and a tool called ESpec) for early requirement and specification descriptions and testing. The core idea behind early testable requirements is that the problem is described before we search for a solution and the problem description drives the design.

The method follows the single model principle, i.e., design specifications written using expressive mathematical models such as sets, bags, sequences and maps are contracts that are integrated into the program text itself. These tightly integrated specifications allow inconsistencies between code, specifications and requirements to be detected as early as possible and during the lifetime of the code. Customer requirements are described using Fit tables and specification violations (where they occur) are indicated in the Fit tables. The method does not depend on a particular code development methodology (e.g. Agile vs. Conventional) and can be used whatever development methodology is preferred.

Acknowledgements

This thesis made possible by the collaborative inputs and contributions of all members at the Software Engineering Lab at York University. Firstly, I would like to thank my supervisor and mentor, Dr. Jonathan Ostroff, for his continuous support throughout this work. His dedication, helpfulness and expertise made this thesis possible. He encouraged me and helped me to get involved in many aspects of an academic life and taught me many valuable lessons for a future in research. Special thanks to Dr. Vassilios Tzerpos for serving as member of my thesis committee and for serving on my oral examining committee.

I'd like to express my gratitude to York University and the Ontario Graduate Scholarship (OGS) program for their continuous financial support throughout my studies.

I'd also like to thank my fellow students Jackie Wang, Eric Kerfoot and David Makalsky for helping me when necessary. I deeply appreciate the help we have received from David Crocker of Escher Technologies for understanding the Perfect Developer and help of Dr. Wolfram Schulte, and Dr. Rustan Leino of Microsoft Research for understanding Spec#. I'd like to thank Peter Gummer and Emmanuel Stapf for their feedback on ESpec. Likewise, I would like to acknowledge helpful feedback from Richard Paige of University of York in England and Bertrand Meyer, Bernd Schoelle and Andreas Leitner of ETH Zurich.

And, last, but not least, I would like to thank all my family members, specially my mother, Fahimeh, and father, Reza, for their non-stop love and support, and for all the sacrifices they had to make to get me to this point. Finally, I thank my wise Bita for her unconditional love, patience and support.

Table of Contents

List of Tables

List of Figures

1 Introduction

The primary measure of success of a software system is the degree to which it meets the purpose for which it was intended. Requirements engineering is the process of discovering that purpose, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation [69].

There are a number of inherent difficulties in eliciting and communicating the requirements. There may be many stakeholders (paying customers, users and developers) with varying goals that may conflict. These goals may not be explicit; the goals may be difficult to articulate and even vague or ambiguous. Inevitably, satisfaction of these goals may be constrained by a variety of factors outside their control [69]. As Fred Brooks wrote, we are dealing with a very difficult problem:

> The hardest part of building a system is deciding what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all interfaces to people, to machines and to other software systems. No other part is more difficult to rectify later [40].

Surveys by the Standish Group [86] appear to show that 23% of all software projects fail before they are completed. Of the remaining projects that are completed, 49% are significantly late, over budget, or do not include all the essential features and the requirements. Only 28% of software projects succeeded, i.e., ship on time, within budget, and with all of the requested features [45]. Although the precision of these statistics have been challenged [44], nobody doubts that the problems are significant [43].

The most significant factors responsible for the lack of success are lack of customer input, incomplete requirements and specifications and changing requirements and specifications. In their recipe for success the Standish group recommends that stakeholders develop the ability to "clearly articulate requirements" and translate these requirements between the business people (the customer) and the technical people (software developers) [45].

It is generally not an easy task for a software developer to write and communicate the requirements. Both formal [69] and informal methods for doing so have been developed, but as one IT specialist wrote:

> I was once in a meeting in which a team had to review a business specification for an application enhancement. The meeting had been scheduled for one hour. It lasted for three painful hours, because the team was stumbling over each paragraph: Verbosity, ambiguity and an avalanche of bullets conspired to hide the meaning of those phrases...
>
> UML might be king in academic circles, but English is still the preferred and most-used tool in the field when it comes to communica-

> tion between business users and developers. I have recently heard a tool vendor trying to score points for his product based on the fact that the product uses plain English, not UML, in order to capture requirements [4].

Writing a good requirements document is difficult in the eyes of most developers. It delays getting on with the coding and it is thus considered a waste of time. But ignoring the requirements is a recipe for disaster (as pointed out by Brooks)—nothing is more difficult to rectify than building the wrong product (the one your customer does not want).

As discussed by Berry *et. al.* [14] there are a number of reasons that writing a *requirements document* for a computer-based system before implementing it is a good idea:

1. The process of writing the requirements document of the system under construction is a good way to learn its requirements and to make it clear what must be implemented to obtain the required system.

2. The process of writing the requirements document of the system helps to reconcile differences among the stakeholders.

3. The requirements document allows the customers of the system to validate that the projected system will be what they want before resources are spent implementing a possibly incorrect system.

4. The requirements document allows deriving both covering test cases and expected results that allow to verify that the implementation of the system does what it is supposed to do.

Despite the clear benefits of writing the requirements document before coding the system, many projects find themselves unable to produce the requirements document for a variety of reasons, some technical and some social [14]:

1. It is difficult to write a good requirements document, one that specifies exactly what the system under construction is supposed to do without limiting unnecessarily how to implement it (i.e., it is hard to specify *what* rather than *how* the system must perform).

2. "Participants in most projects these days believe that they do not have the time to do so, that it is necessary to proceed immediately, if not before, to coding, in order to meet the codes delivery deadline or to be the first in the market with the codes functionality (begging the question of how do they know what to implement anyway if requirements are not specified)." [14]

3. Participants in most projects these days perceive that time spent on writing requirements document is wasted since the requirements will change anyway and the requirements may never be read, even by the implementers.

The authors of [14] suggest that a *user manual* makes an excellent software requirements document. The method produces a document that delivers the benefits of writing a requirement document before implementation (enumerated earlier) and helps mitigate the three problems that discourage the production of requirements document before implementation.

1.1 Thesis Motivation

Suppose we think of a requirements document as a user's manual (or alternatively, as a document containing a mix of English text, descriptions of user interfaces and informal sketches). Now, a user's manual cannot be directly tested (although it may be used as the basis for developing tests). Is there a way to make a requirements document such as a user's manual directly testable? In this thesis, we will investigate the use of the Fit framework [31] to do just this by adding some additional information in a notation that is "user-friendly", i.e., understandable by our customers and mechanically testable with the right kind of tools. The idea of specifying before implementing and specifying in a testable way will be used throughout the proposals in this thesis from requirements to the final code.

In this thesis, *customer requirements* (in the problem domain) are differentiated from *design specifications* (in the solution space). A design specification is the artifact intermediate between implemented code and the customer requirements. We argue that customer requirements and design specifications should be testable and testable early in the design cycle leading to early detection of requirement and specification errors. We thus provide a method (and a tool called ESpec for Eiffel) for early requirement and specification descriptions and testing. The core idea behind early testable requirements is that the problem is described before we search for a solution and the problem description drives the design.

The method follows the single model principle [76], i.e., design specifications written using expressive mathematical models such as sets, bags, sequences and maps are contracts that are integrated into the program text itself. These tightly integrated specifications allow inconsistencies between code, specifications and requirements to be detected as early as possible and during the lifetime of the code. Customer requirements are described using Fit tables and specification violations (where they occur) are indicated in the Fit tables. The method does not depend on a particular code development methodology (e.g. Agile vs. Conventional) and can be used whatever development methodology is preferred.

This method is presented in the Eiffel language [66] which has a mature contracting mechanism, but the conceptual ideas could be used in any of the emerging contracting languages such as Spec# [9] and ESC/Java [59].

1.2 Requirements vs. Specifications

According to Jackson [54], a software application such as a word processor is a machine—one similar to a typewriter, but with more versatility. Similarly, a software telephone switch is a machine—one similar to an old-fashioned telephone exchange, except that the new kind of machine does not consist of rotary switches and clattering relays. The purpose of software development is to build special kinds of machines. A general purpose computer accepts our description of the particular machine that we want (as described in the code), and converts itself into the desired machine.

1.2.1 The Machine Domain and the Problem Domain

The purpose of the machine (such as a software telephone switch) is for it to interact with and achieve some effect in the world (e.g., help people make phone calls). The part of the world in which the machine's effects will be felt—and which is of most interest to customers of the machine—is called the problem domain, which we denote by the letter P. It is always right to pay serious attention to the problem domain. We let the letter C stand for the machine (i.e., the implemented code).

If we are developing a program to control an airplane, we obviously need to understand how the airplane works, how it lands and takes off on runways, and how it can be controlled while in the air. We may also need to understand intangibles associated with the problem domain, such as the rules for safe aviation. This understanding must be made prior to any attempt to lay out the data structures and data flow of the code that will ultimately control the airplane. The phenomena (states and events) of the problem domain are clearly distinct from the phenomena of the machine domain (code and data structures) required to operate it.

1.2.2 Requirements

The phenomena of the problem domain determine the customer's Requirements (R). Requirements are the goals of the customer expressed in terms of the phenomena of the problem domain. Software requirements may be expressed in various formal notations (e.g., predicate logic) or semi-formal notations such as UML (e.g., use cases). But, more often than not, requirements are expressed using a combination of English text, user interface drawings and rough sketches. As mentioned earlier, a well-written user's manual is a type of requirements document.

Requirements are therefore about the phenomena of the problem domain P and not about the phenomena of the code C. Not all the phenomena of the problem domain are necessarily shared with the code. But, the code does share some phenomena with the problem domain. The code can try to ensure that the requirements are satisfied by manipulating the shared phenomena at the interface of P and C. An example of a shared phenomenon is the event of a passenger sitting in an aircraft seat and pushing a button to turn on a light. The push of the button is a phenomenon that is shared at the interface between the passenger (in the problem domain) and the control software (in the machine domain). To the passenger the event is "push the button", and to the machine the event might be "input signal on interrupt line L1".

1.2.3 Specifications

As we mentioned, not all of the phenomena of the problem domain are shared with the code. There can thus be a gap between the customer's requirements and what the code can deliver directly. A *Specification* (S) is a bridge between the phenomena of the problem domain and the phenomena of the code, describing phenomena (inputs and outputs) at the intersection of P and C.

A specification in this context is a precise mathematical description of some desired unit of functionality of the product. There are many types of system specification which are written in different languages (e.g., formal languages like B [1]). A specification is the developer's *model* of the software product under con-

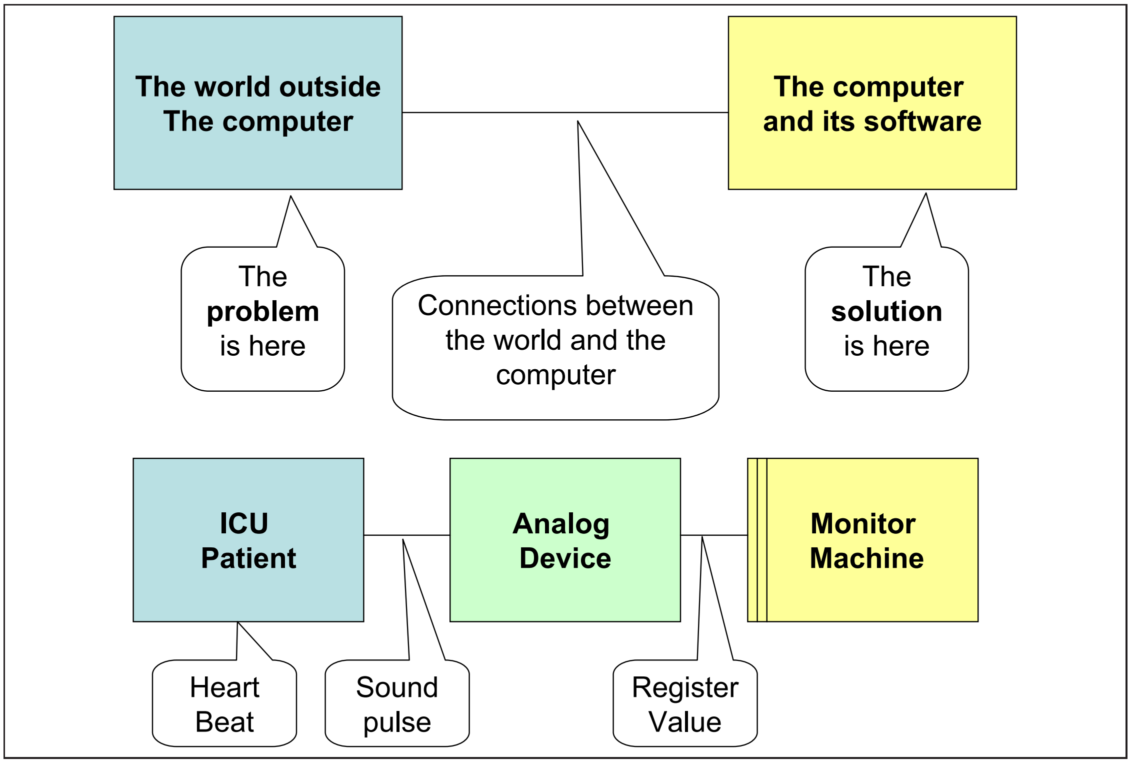


Figure 1.1: The Machine and the Problem Domain

struction, akin to an engineer's blueprint and can be expressed in several ways, such as a *contract* between the supplier and the user of the product. Although a specification may itself be (a high level) program text and may also be executable, it is not the same thing as the final code for a software product. Rather, it is an abstraction of the program under development, which allows us to reason about the program during construction.

1.2.4 An Example

Consider the diagram in Fig. 1.1 illustrating the problem of measuring vital signs such as the heartbeat of a patient in an ICU taken from [55]. There are four different descriptions of the patient monitoring system:

P—Problem Domain: A patient's heart can beat from 0 to 170 beats/Sec (predetermined by human physiology).

R—Requirement: Monitor the patient's heart beat and sound an alarm if it is outside of the range from 60 to 100 beats per minute.

S—Specification: `Alarm-Register := False` when the Sound-Pulse-Register is outside the range hexadecimal 3C to hexadecimal 64.

C—Computer Code: The machine code that implements specification **S**.

The central requirement *R* is to monitor the heartbeat—not the sound pulses or the register values in the machine (i.e., the implemented computer code). The requirements are the effects in the problem domain that your customer wants the machine to guarantee. The requirements are all about the phenomena of the problem domain (not the machine). The predicate *P* described the fixed constraints emerging from the problem domain.

The specification S refers to phenomena shared by the problem domain and the machine. S specifies a design solution that we hope to satisfy the requirements R. Finally, C is a description of the computer code needed to implement the design specification S.

1.3 Rational Development Process

Our core idea behind early testable requirements and specification is as follows: Requirements should be testable as early as possible so that the problem is stated before we search for a solution. We also want the problem to drive the design. A rational software development might proceed as follows:

- Elicit and document the Requirements R of the customer in terms of the phenomena in the problem domain. Constraints of the problem domain are described by P.
- From the Requirements, derive a Specification S for the software code that must be developed.
- From the Specification, derive a machine C (the code).

We may describe the development process as follows:

1. **Specification correctness:** $P \wedge S \rightarrow R$
2. **Implementation correctness:** $C \rightarrow S$
3. **System correctness:** From (1) and (2) conclude that: $P \wedge C \rightarrow R$

The first equation asserts that any behaviour of the system that satisfies the specification S in the problem domain P also satisfies the requirements R. Equation (1) is called *specification correctness* because it says that the solution S specified by the developer will satisfy the customer's goals as expressed in R (i.e., we are developing the right product—the one desired by the customer as described by R).

The second equation asserts that any behaviour executed by the implemented code C satisfies the specification S. This means that the software product is correct and we thus call equation (2) *implementation correctness*. Equation (1) checks that we are developing the right product (often called validation) and (2) checks that we are developing the product right (often called verification).

The third formula, which is a consequence of formulas (1) and (2), asserts that our implemented solution C in the problem domain P satisfies the customer requirements R.

It is important to note that we do not pose any obligations on the developers to follow a strict methodology:

> Anyone who has been involved in intellectually taxing activities trying to understand and solve a complex problem knows that the process of arriving at a good solution is far from regular. On the contrary, the most common impression during the course of the effort is often a sense of total disorder and utter confusion. This is also true for cases where the final result eventually turns out to be very simple and elegant, and the greatest sense of confusion is often experienced shortly before the crucial perspective is discovered. So the bad news is that a rational process, where each step follows logically from the previous ones and everything is done in the most economic order, does not exist. Complex problem solving just does not work that way. But the good news is that we can fake it. We can try to follow an established procedure as closely as possible, and when we finally have our solution (achieved as usual through numerous departures from the ideal process), we can produce the documentation that would have resulted if we had followed the ideal process. [79]

This gives us a number of advantages. (a) The process will guide us, even if we do not always follow it. When we are overwhelmed by the complexity of a task, it can give us a good idea about how to proceed. (b) We will come closer to rational modeling if we try to follow a reasonable procedure instead of just working *ad hoc*. (c) It also becomes easier to measure progress. We can compare what has been produced to what the ideal process calls for, and identify areas where we are behind (or ahead).

1.3.1 Testability and Tool Support for Testability

A key idea in this thesis is that Requirements and Specifications should be testable. The distinction between a Requirement and a Specification (see previous section) may now be used to define what we mean by testable Requirements and Specifications.

What are testable requirements? Formula (1) in the previous section asserted the following relationship between a specification S and a requirement R: $P \wedge S \rightarrow R$. To test requirement R is to check that the suggested solution S entails the requirement. If the requirements are described informally (e.g., as English text) then there is no real way to test them mechanically. Furthermore, as the solution S is refined, we will want to check $P \wedge S \rightarrow R$ repeatedly (as in continuous regression testing). So it would be advantageous to mechanize requirement testing. This means that both specifications and requirements must be formalized in order to mechanize requirement testing.

Our approach will be to use Fit tables to formalize requirements in a language understandable to customers (i.e., it is not a programming language). Specifications will be formalized either as ML-Contracts or as Scenario Tests as will be explained in the sequel.

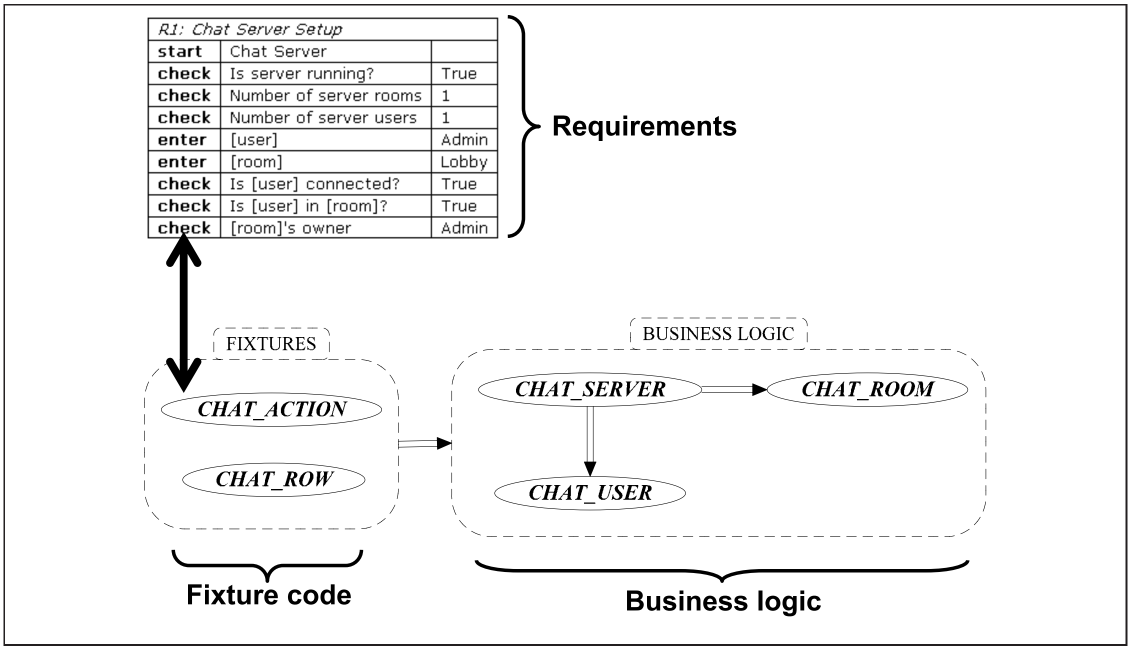
What are testable specifications? Formula (2) in the previous section asserted the following relationship between implemented code C and the specification S: $C \rightarrow S$. To test the specification S is to verify that the implemented code satisfies the specification. If the specification is an ML-Contract, we may do this verification using either run-time assertion checking or formal theorem proving. If the specification is a Scenario Test, then the check can be performed by executing the code and checking that the results specified in the Scenario Test are achieved. Thus, we can also mechanize specification testing.

1.3.2 The ESpec Tool for Testability

The tool support developed as part of this thesis is called *ESpec*. The purpose of ESpec is to provide mechanized support throughout the software development process for writing and testing customer requirements and design specifications. ESpec itself consists of three components: *ES-Fit*, *ES-Test* and *ES-Verify*.

Consider the diagram in Fig. 1.2 which provides an example of a Fit table (labeled "Requirements"). This table describes a scenario provided by the customer as a sequence of actions and checks that must hold after these actions are taken. The requirements document (e.g., a user manual) can be decorated with such tables which then become testable.

How does the developer satisfy the requirements specified in the customer-provided Fit table? The developer will need to write two kinds of classes: *Fixture*



R1: Chat Server Setup		
start	Chat Server	
check	Is server running?	True
check	Number of server rooms	1
check	Number of server users	1
enter	[user]	Admin
enter	[room]	Lobby
check	Is [user] connected?	True
check	Is [user] in [room]?	True
check	[room]'s owner	Admin

Figure 1.2: Relationship between Fit tables, Fixtures and the Business logic

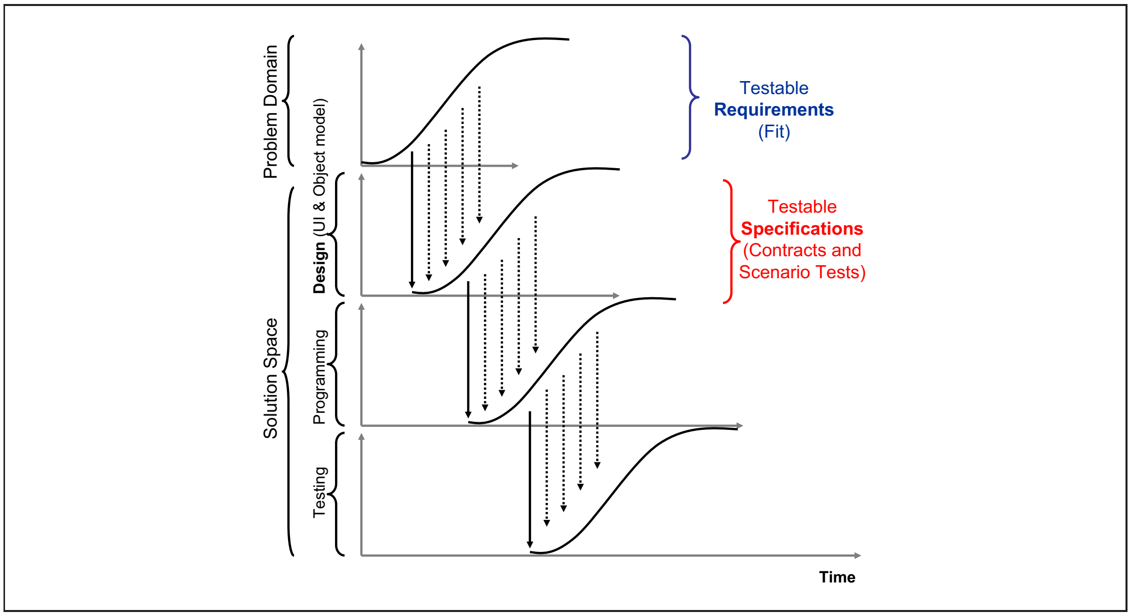


Figure 1.3: Early Testable Requirements/Specifications in Development-cycle

classes and classes of the business logic (see Fig. 1.2). Fixtures are glue code between the customer-provided requirements and the business logic *C*. ES-Fit provides libraries that allow the developer to easily develop such Fixtures that connect the requirements to the business logic. ES-Fit uses the developer written Fixture classes to parse the requirement document, extract the tables, interpret the tables and invoke the relevant business logic and then reflect the results of running the business logic back to the tables in the requirements document. The rows in tables where the checks succeed are coloured green and those that fail are coloured red.

Fig. 1.3 shows the design cycle starting with requirements elicitation which is followed by design, coding and then (conventionally) testing. Fit tables can of course be developed by customers early in the process, before design and coding activities. As the design proceeds, the implemented code can be continuously tested against the behaviour described in the Fit tables.

The ES-Test component of the ESpec tool is used to check that the implemented code satisfies the design specifications. Specifications are written using Scenario Tests and ML-Contracts.

In Test Driven Development [10], unit tests are themselves executable code that check the correctness of a "unit" of some module such as a method. The critical insights are (a) that the test can be written before the method implementation, and (b) that the test is a specification of the method. By forcing developers to specify the functionality up-front, the description becomes more abstract (focussed more on what the feature should do rather than how it should do it) and thus closer to a specification. Scenario Tests use the same framework as unit tests but specify the interactions or collaborations among various modules (classes) of the system to achieve some unit of functionality. Like unit tests, Scenario Tests can be written before any class implementations. As the design proceeds and the code is produced, the implemented code can be tested continuously to ensure that the Scenario Tests (and any unit tests) are satisfied.

Design by Contract (DbC) [64] is a way of specifying the mutual obligations and benefits of clients and suppliers of classes. DbC is an important part of Eiffel [66], and it is also supported in UML [38] via OCL [26]. The standard Eiffel DbC contracts do not have the full mathematical power of OCL. Thus, for specifying contracts, we use ML-Contracts [72] developed by a team that included this author. ML is a mathematical modeling library written in Eiffel that uses mathematical sets, bags, sequences and maps in contracts that are integrated into the program text itself. These tightly integrated specifications allow inconsistencies between code, specifications and requirements to be detected as early as possible and during the lifetime of the code. As defined in the OMG standard [70], OCL is a description language and no executable semantics is supplied. Thus, OCL cannot be used to test code (say Java) without third party add-ons. By contrast, the ML library is executable code written in Eiffel. As a result, implemented code can be tested against the specifications written as ML-Contracts. ML-Contracts satisfy the single model principle [76], i.e., the contracts are part of the program text.

ES-Test checks ML-Contracts via runtime assertion checking. However, as discussed in [72], Eiffel code with embedded ML-Contracts can also be verified using a theorem prover. The ES-Verify component of ESpec may be used to run the theorem proving tools developed in [72].

Using ESpec, testable requirements and specifications can be written and checked under a single green/red bar as will be explained in the sequel.

1.4 Organization of this thesis

This thesis is organized as follows:

- Chapter 1 is an introduction that presents the background, motivation and contribution for the method and the tool presented in this thesis.

- Chapter 2 discusses the idea of Testable Specifications by introducing various types of system specifications such as ML-Contracts and Scenario Tests that we use in our method. This chapter provides an overview on how

ESpec's Unit Testing framework (ES-Test) can be used to capture and test these specifications throughout the development process.

- Chapter 3 presents the idea of Testable Requirements by giving an overview of the Fit framework and various types of tests supported by this framework. We also go over various kinds of Fixture code (the glue code between the customer-provided Fit tables and the system under test). With simple examples we show how Testable Requirements can be used to detect specification and/or implementation errors in the underlying system.

- Chapter 4 illustrates our method of Early Testable Requirements and Specifications with a case study (Chat room example). This example shows how specification and implementation bugs can be detected throughout the development process.

- Chapter 5 is mainly devoted to the design and implementation of ESpec tool itself. We describe the challenges and our design decisions in developing ES-Fit and integrating various components of the ESpec tool such as ES-Test, ES-Verify and Mathematical Library (ML).

- Chapter 6 discusses the related research and compares this work to others.

- Chapter 7 concludes the thesis.

- The Appendix contain a brief introduction to Eiffel language and agent mechanism (Appendix A) the code for the example discussed in Chapter 4 (Appendix B), a discussion on ES-Verify (Appendix C). This component of the tool translates Eiffel code with ML-Contracts into the Perfect Developer specification language. The generated code may then be verified using the Perfect Developer automatic theorem prover. Feedback from the theorem prover is reflected back into the ESpec tool. Finally, the ESpec tool user′s manual and the screen shots are provided in Appendix D.

1.5 Research contributions

The method and ESpec tool reported in this thesis was the basis of an invited contribution to the *Tests and Proofs* Conference (TAP'07) in Zurich [74]. The contributions are listed below.

1.5.1 Fit framework for Eiffel

The Fit framework of [68] may be used to write testable customer requirements prior to (or during) the code development. This thesis provides the first implementation of Fit adapted to and extended for Eiffel.

1.5.2 Extensions to the Fit framework

The original Fit framework of [68] is written in Java. Fit tests are run from the command line. The Fit framework developed in this thesis (called ES-Fit) is an Eiffel library for writing Fit Fixtures (the glue code between the Fit tables and the system under test) and a convenient graphical tool for editing and running Fit tables and displaying the results of Fit tests. For the convenience of customers and developers, ES-Fit extends the standard framework with new constructs which appear as keywords in Fit tables. For example, the keyword **reference** is used by a customer to describe a Fit table that acts as a global database of values that may be queried from any other Fit table. The fixture library is designed to provide flexible constructs for defining new fixture types.

1.5.3 Integrating ML-Contracts into Fit tables

Eiffel has a built-in Design by Contract (DbC) mechanism. A further contribution is that ES-Fit ensures that contract violations are reported directly in Fit tables. This allows the customer to observe and report these violations allowing customers to provide early and specific feedback to developers.

The built-in Eiffel DbC mechanism is incomplete. It does not by itself allow for complete abstract mathematical specifications. For example, a deferred class for a stack has no implementation and thus the contract for the push operation (for example) would be incomplete. The authors of [72] (including this author) developed an expressive executable mathematical library in Eiffel called ML based on sets, bags, sequences and maps. With ML, complete contracts based on mathematical models become possible obeying the single model principle [76] (i.e., the mathematical contracts are part of the program text). Since the contracts are executable, implementations can be checked against the code at runtime via assertion checking. As described in [72], ML may be translated into the specification language of the Perfect Developer theorem prover and implementations can be mechanically verified against the contracts for a subset of Eiffel. ML is part of the ESpec software quality workbench. ESpec tool is described below.

1.5.4 ES-Test improvements

ES-Test is a successor to an Eiffel unit testing library called E-Tester reported in [73]. The contribution of this author to further development of the tool includes:

- Allowing unit tests to be integrated with other checks (such as Fit tests and static verification) in a single test suite.
- The addition of Tagged Violation Tests (see section 2.5.2).
- Complete reports of the type of contract violation and the tag involved in

the violation (both in the GUI and the command line version of the tool), e.g., a developer may use the `show_error` command to reflect the complete call stack in the report (see Chapter 5).

1.5.5 ESpec tool: software quality workbench

- The ESpec tool is another contribution of this thesis that was developed to support testable requirements and design specifications in an integrated framework consisting of three components: ES-Fit, ES-Test and ES-Verify. The tool allows the developer to write a test suite that consists of Fit fixtures for running Fit tables (executed by ES-Fit), Scenario and unit tests (executed by ES-Test) and code verification against ML-Contracts (executed by ES-Verify).

  The design in Fig. 1.4 shows how a developer can integrate Fit table checks (for requirements) and specification checks within a single test suite. Eiffel's multiple inheritance capability [67] was useful in this respect as a test suite (`ES_SUITE`) inherits ES-Fit, ES-Test and ES-Verify capabilities simultaneously.

- In Chapter 4 of the thesis, a chat application is developed using Fit tables (for testable customer requirements) and ML-Contracts and Scenario Test

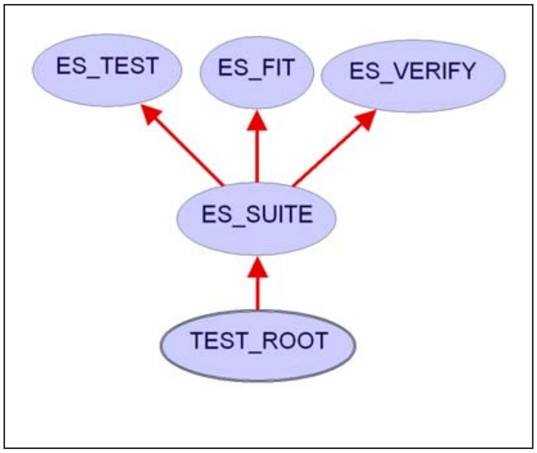


Figure 1.4: ES_SUITE architecture

(for testable design specifications). This chapter illustrates the integrated use of the method and tool for early testable requirements and specification.

- A Scenario Test involves a collaboration between a number of classes to achieve some specified result that emerges through the collaboration. ES-Test will report contract violations during the collaboration (if they fail) as well as failure to achieve the specified result. ESpec aggregates all tests under a single green/red bar to report overall success or failure.

1.5.6 Detection of specification and code errors

Every time the ESpec tool is invoked all the requirement and specification tests are executed. Two types of errors may be reported by the tool. A *category one* error occurs when a customer requirement (in a Fit Table) is checked against the implemented code and the actual result generated by the code does not match the customer's expectation (no contract violation is reported, just a mismatch in expectation). Such an error may be an indication of a specification problem because a properly specified design should have generated a contract violation.

A *category two* error is any contract violation whether it is reported in a Fit table or elsewhere. Such a violation may indicate an implementation problem in the code (the code does not satisfy the design specification).

1.5.7 ESpec maintenance and support

ESpec has been used as a mandatory part of a Software Design[1] course at York University since its first release in the Winter of 2005. ESpec is maintained under the GPL licence for public download (see `http://www.cse.yorku.ca/~sel/espec/`). About 3000 downloads have been recorded worldwide. The tool has been mentioned on various Eiffel groups[2].

[1] AK/CSE 3311 3.00 Software Design: A study of design methods and their use in the correct implementation, maintenance and evolution of software systems. Topics include design, implementation, testing, documentation needs and standards, support tools. Students design and implement components of a software system.

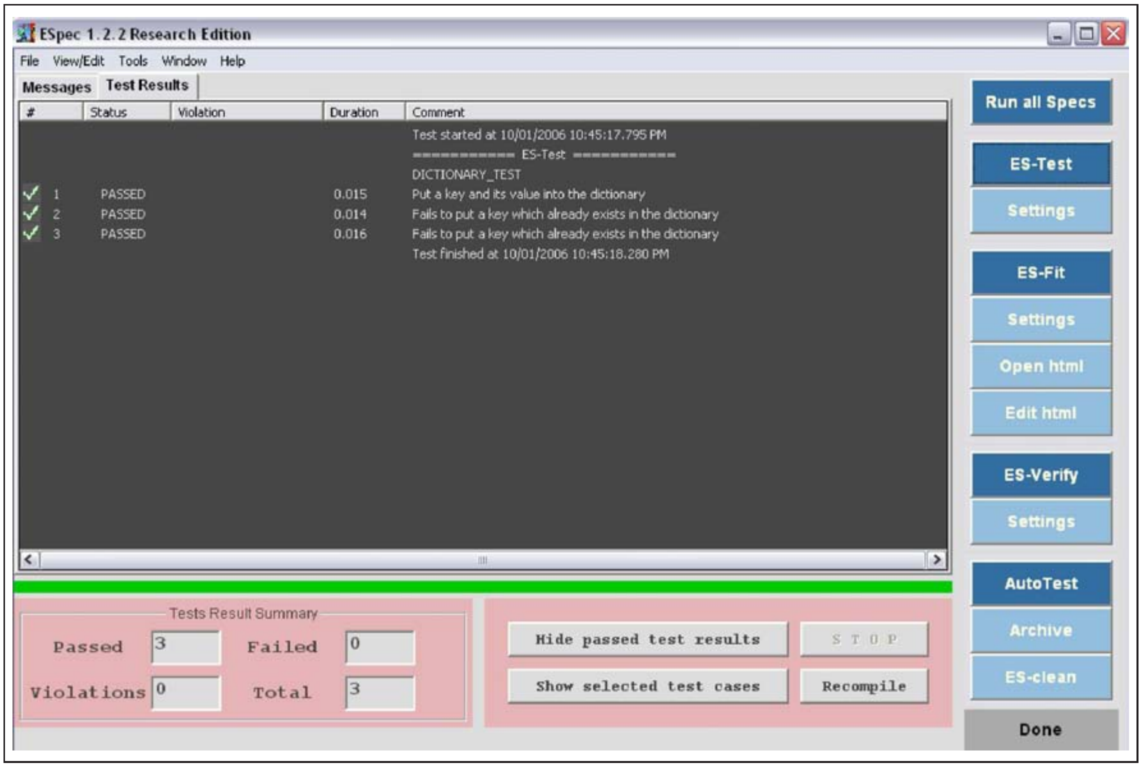
[2] e.g., `teameiffel.blogspot.com/2006/11/eiffel-specification-package-updated_15.html`

2 Testable Specifications

In the previous chapter, we distinguished between Requirements and Specifications (see Section 1.2). In this chapter, we explore two different types of Specifications: *ML-Contracts* [72] and *Scenario Tests*.

ML-Contract and Scenario Test specifications are "testable" in the sense that the underlying code implementation can easily be checked against them. These specifications may be written early, i.e., they may be written before the implemented code is developed. As discussed in Section 1.1, early specifications are not enforced by our method and tool; thus, specifications may be written at any point in the development. However, the earlier they are written, the earlier they can be used by developers to detect bugs in the development process. We show how our software quality workbench, ESpec, is used as an integral part of the development process.

The technique (and accompanying ESpec tool) use Eiffel programming language [66] (and its UML-like modeling language, BON [75]). However, these techniques are not limited to Eiffel and can be used with any language that has a suitably expressive contracting mechanism (e.g., ESC/Java2 [59] and Spec# [8]). A snapshot of ESpec GUI showing the use of the ES-Test component is shown in Fig. 2.1.



#	Status	Violation	Duration	Comment
				Test started at 10/01/2006 10:45:17.795 PM
				=========== ES-Test ===========
				DICTIONARY_TEST
✓ 1	PASSED		0.015	Put a key and its value into the dictionary
✓ 2	PASSED		0.014	Fails to put a key which already exists in the dictionary
✓ 3	PASSED		0.016	Fails to put a key which already exists in the dictionary
				Test finished at 10/01/2006 10:45:18.280 PM

Tests Result Summary

Passed	3	Failed	0
Violations	0	Total	3

Figure 2.1: Snapshot of ESpec GUI after execution of ES-Test

As was described in the first chapter, specifications are descriptions of properties of interest about a system. When developers have a clear understanding of the requirements, they can start to produce system specifications. There are many types of system specifications which are written in different languages (e.g., formal languages like B [1]). Each type of system specification has a different characteristic and serves a different purpose.

However, in general, a specification describes a property of the code (of the program, or machine, under construction) that is an abstraction of the program which also allows us to reason about it during its construction. For example, the specification of a routine to `reverse` a list of items does not explain *how* to reverse elements in the list but *what* the reverse routine does. A specification of `reverse` may, for example, express the relationship between the initial and final states of the list. We can then reason about the list. For example, if the `reverse` routine has been specified correctly, then reversing the list and reversing it again should yield the original list. This activity may take place long before the body of the routine is implemented.

Assuming that we have a notation for expressing specifications, then the formula $C \rightarrow S$ expresses the desired relationship between a specification S of a system and the implemented code C for that system (see implementation correctness in Section 1.3). This relationship can be used to test that the specification is satisfied, i.e., to check that system behaviors as generated by the code C satisfy the specification S.

Such tests of specifications could be done manually. However, it would be a tedious task and prone to human error. It is desirable that the checks for correctness be done mechanically where possible. For mechanized checking we obviously require that the specification notation be amenable to mechanized testing. Both ML-Contract and Scenario Test specifications can be checked mechanically as we explain in the following sections.

2.1 Why double the work?

Our proposal is that developers (a) write machine checkable specifications and (b) then implement the code. It is reasonable to ask: "does this approach not unnecessarily double the work of the developer?". Why not just proceed to implementation code once the informal specification is known? Why this extra burden on the developer?

To answer this question, we may refer to the "second time phenomenon" discussed by Daniel Berry [13] in the context of requirements engineering:

> In 1985, I published a paper with Jeannette Wing that suggests that FMs [Formal Methods] work, not because of any inherent property of FMs as opposed to just plain programming, which is really also an FM, but rather, because of the second time phenomenon [16]. If you do anything a second time around you do better, because you have learned from your mistakes the first time around. Indeed, Fred Brooks says: Plan to throw one [the first one] away; you will anyway! [11]. In other words, you cannot get it right until the second time. If you write a formal specification and then you write code, you've done the problem formally two times.

Berry explains that informal specifications will not have the same effect as a machine checkable formal specification. In an informal specification it is too easy to overlook details thus leading to failure.

> Note that writing an informal specification and then writing code does not have the same effect. It is too easy to handwave and overlook details and thus fail to find the mistakes from which you learn. It has to be two formal developments, specifications or code, for the second-time phenomenon to work. Observe how the two-time approach is requirements centered. One is not going to fix implementation errors this way, because the second time is not the same implementation as the first time. Even if they were the same, one can introduce new errors in the rewrite. The focus of the first specification or coding effort is on understanding the essence and eliminating requirements errors. The focus of the second is on implementing the understood essence. As Euripedes says, Second thoughts are always wiser.

Thus, it is worthwhile doing both a formal specification (in the sense mentioned by Berry) as well as the implementation code. Specifications help us understand the essence of what must be built by eliminating requirement errors. Since the specifications are machine checkable we cannot just handwave and overlook details that lead to a mistaken view of the product that must be built.

2.2 Contracts as Specifications

Contracts may be used to specify the behavior of classes (or modules). Each feature of a class is provided with a precondition and postcondition, and class invariants specify global constraints on the data structures of the class.

Contracts specify the obligations and benefits between the user (or client) of a module and the developer (or supplier) of the module. Clients may invoke the module if the precondition is satisfied and the supplier must guarantee the postcondition. As in human affairs, a good contract brings with it obligations as well as benefits for both parties—with an obligation for one usually turning into a benefit for the other. This is also true of contracts between classes [67]:

- A **precondition** binds the client: it defines the conditions under which a call to the routine is legitimate. It is an obligation for the client and a benefit for the supplier.

- A **postcondition** binds the supplier: it defines the conditions that must be ensured by the routine on return. It is a benefit for the client and an obligation for the supplier.

For precision, specifications of software components are usually written using predicate logic and set theory as contracts between the supplier of the component and the users. For testability, specifications must be embedded in program text, and amenable to compiler checks such as type checking and static analysis of program properties (e.g., null pointer de-referencing) [76]. A pioneering approach to writing testable specifications of this kind was the use of Design by Contract (DbC) in Eiffel.

DbC [65] is a lightweight formal technique for engineering software systems with significant requirements for reliability and robustness. It integrates mathematical descriptions with code, ensuring consistency, and it is designed to be supported by tools that are comfortable and familiar to developers, e.g., compilers, debuggers, static checkers, and testing frameworks.

The term "Design by Contract" was coined by Bertrand Meyer in connection with his design of the Eiffel programming language and first described in various articles starting in the mid 1980s (e.g., [64]) and the two successive editions (1997, 1998) of his book Object-Oriented Software Construction [67]. Design by Contract has its root in work on formal verification, formal specification and Hoare logic [49].

The contracting language has many benefits. (a) Contracts are precise specifications of the required module behaviour. (b) Contracts document the class API (application program interface) for both clients and suppliers. (c) Contracts can be tested (at run-time) or verified (at compile time). (d) Contracts can be used to define the notion of an exception (behaviour that violates the contract). (e)

Contracts appropriately constrain redefinitions of methods in descendant classes (Liskov substitution principle [62]). The DbC approach has been extended to Java (e.g., JML [18]) and C# (e.g., Spec# [8]), and thus can be used effectively with these languages as well.

2.2.1 Basic Contracts

```
class
    SORTABLE_ARRAY[G -> COMPARABLE]
inherit
    ARRAY[G]

feature -- commands

    sort is
             -- sort the array
        require
            $\forall i : INTEGER \mid valid\_index(i) \bullet item(i) \neq Void$
        do
            -- an algorithm
        ensure
            $\forall i : INTEGER \mid valid\_index(i) \wedge i < count \bullet item(i) \leq item(i+1)$
            permutation (Current, old Current.twin)
        end

feature -- queries

    permutation (initial; final: ARRAY[G]): BOOLEAN is
        -- Is the array ‘final’ a permutation of ‘initial’ ...

invariant
    count > 0

end
```

Figure 2.2: Specification of a sorted array

Consider the specification of a class SORTABLE_ARRAY in generic parameter G as shown in Fig. 2.2. The generic parameter `G` is constrained to inherit from class `COMPARABLE`[3]. So we can have an array of `INTEGER`, `REAL`, `STRING` or any suitably compatible user defined classes.

The class `SORTABLE_ARRAY` inherits features such as `count` (the number of items in the array) and `item(i)` (which is the element at index `i`) from class `ARRAY`. The invariant `count > 0` asserts that sortable arrays contain at least one element. The invariant is established by the creation routine and must be true before and after any subsequent routine calls (including new routines such as `sort` as well as routines inherited from class `ARRAY`). The precondition asserts (in the BON mathematical notation) that:

$$(\forall i : INTEGER \mid valid\_index(i) \bullet item(i) \neq Void) \tag{2.1}$$

i.e., there are no void elements in the array (where, $valid\_index(i) \equiv lower \leq i \leq upper$). This property and other predicate quantifiers (e.g., $\exists$) are expressed in Eiffel using agents (see Appendix A for more on agents). The first postcondition asserts that the `sort` routine terminates with the array sorted, and the final postcondition asserts that the final sorted array is a permutation of the initial unsorted array.

Contract specifications follow the single model principle [76]. The class con-

[3] Any class that inherits from `COMPARABLE` comes equipped with a total order.

tracts, consisting of expressive preconditions, postconditions and class invariants are an executable part of the program text and constitute a specification that describe sortable arrays without constraining implementation details and algorithms.

How can the specification be tested? One way is to use a suitable theorem prover (as in Appendix C). We can then formally prove that the implementation entails the specification. Likewise, we may reason about the behaviour of any client that uses the sorted array by using the contracts without the need to know the implementation details. So testability in this case reduces to verification (*theorem proving*).

Alternatively, the specification of sortable arrays can be tested by runtime assertion checking. In this lightweight approach only the executable behaviours (execution paths) invoked by the test suite are checked against the specification. If the contract holds, then no exception is generated. If the contract is violated, either by a client or the supplier not satisfying their respective obligations, then an exception is generated. Thus, in this way, testability translates into executability and the use of exceptions to signal when the contract is broken.

2.2.2 ML-Contracts

The basic Eiffel contracting mechanism is not always sufficient for specifying complex data structures. Abstract classes may not have sufficient implementation detail so that properties can be adequately described. Even where full concrete implementation is supplied, the implementations may be too low level for writing succinct specifications.

In order to write contracts at a much higher level of abstraction, we use the Eiffel Mathematical modeling Library (ML) developed in [72] for specifying the abstract state of a program without exposing its implementation details. This library is similar to the model-based specifications as in B [1] and Z [83], except that it is object-oriented.

The Eiffel ML library contains mathematical collections such as `ML_SEQ`, `ML_SET`, `ML_BAG`, and `ML_MAP`. Fig. 2.3 shows the `ML` class structure. Instances of these classes are both *immutable* and executable. An object is immutable if its state cannot be modified after it is created. This is in contrast to a mutable object, which can be modified after it is created. Other than creation features, all features of the immutable (hence mathematical) ML classes are *queries* (there are no commands). Queries are features that do not change the state of the underlying objects. Rather, queries return values without affecting the state.

A class describes some data structure and the operations that can be invoked on the data. The mathematical structures of the ML library (such as sets and maps) may be used to provide a high-level *model* of the data structure. The operations on the data structure can then be described by contracts written using ML queries with respect to the model. Any subsequent implementation of the operation in terms of efficient mutable classes must satisfy the contracts described in terms of the ML model.

Since ML-Contracts are executable, when runtime assertion checking is turned on, contract violations (if any) are signalled via exceptions, thus indicating an inconsistency between the implementation and its specification. The complete specification of a system and its implementation can be provided in the same compilable and executable Eiffel text. The immutable ML classes will be inefficient (due to its re-construction of a new ML object every time a feature such as `appended_by` is invoked), by comparison to the mutable classes in Eiffel (such as `ARRAY` and `LIST`). But this is acceptable as contract checking may be turned off in the final delivered code which will only use the efficient base library for implementation.

As a simple example, consider the BON contract view of a generic stack as shown in Fig. 2.4a. The model of the stack consists of a `ML_SEQ[G]` (i.e., a sequence of items of type `G`, where `G` is a generic parameter) and `count` (the number of

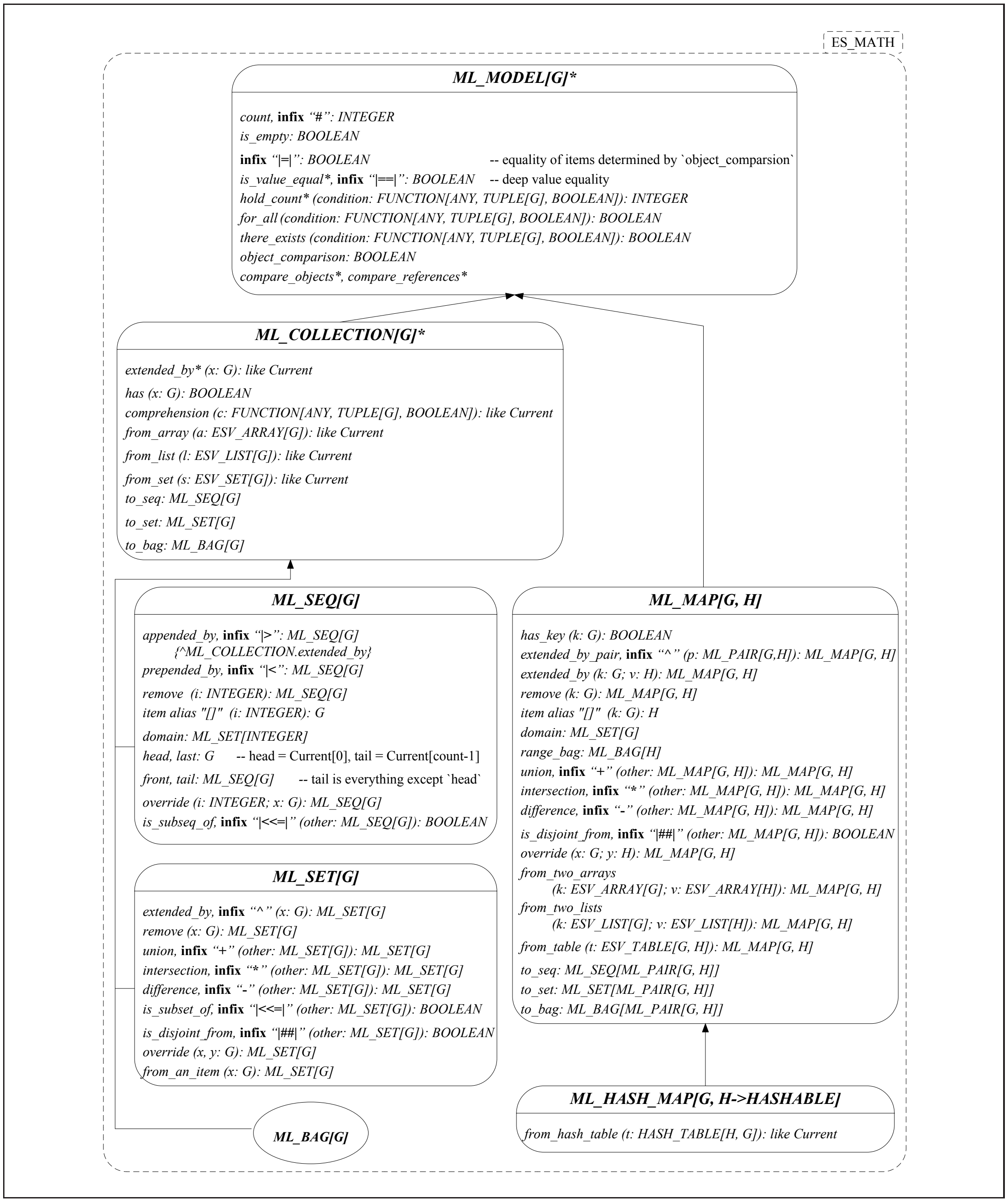
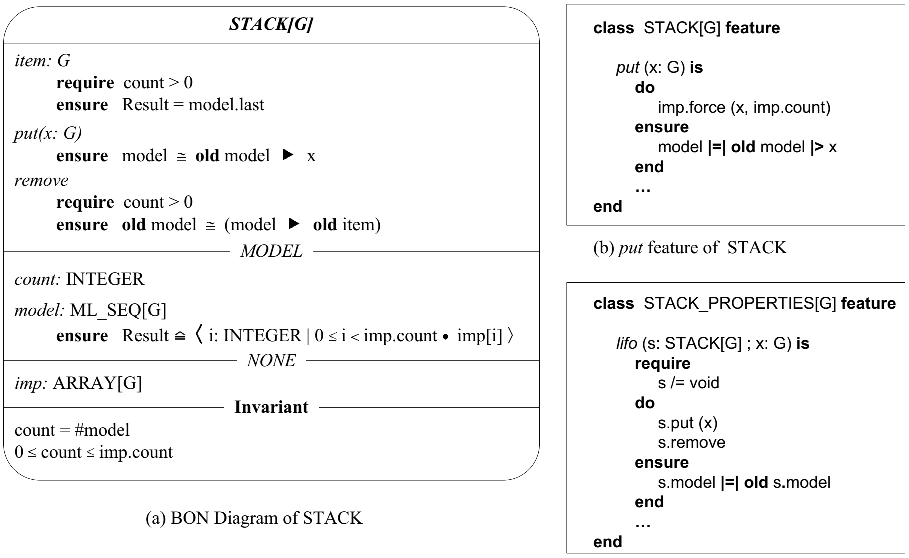

ES_MATH
ML_MODEL[G]*
count, infix “#”: INTEGER
is_empty: BOOLEAN
infix “|=|”: BOOLEAN -- equality of items determined by `object_comparsion`
is_value_equal*, infix “|==|”: BOOLEAN -- deep value equality
hold_count* (condition: FUNCTION[ANY, TUPLE[G], BOOLEAN]): INTEGER
for_all (condition: FUNCTION[ANY, TUPLE[G], BOOLEAN]): BOOLEAN
there_exists (condition: FUNCTION[ANY, TUPLE[G], BOOLEAN]): BOOLEAN
object_comparison: BOOLEAN
compare_objects*, compare_references*
ML_COLLECTION[G]*
extended_by* (x: G): like Current
has (x: G): BOOLEAN
comprehension (c: FUNCTION[ANY, TUPLE[G], BOOLEAN]): like Current
from_array (a: ESV_ARRAY[G]): like Current
from_list (l: ESV_LIST[G]): like Current
from_set (s: ESV_SET[G]): like Current
to_seq: ML_SEQ[G]
to_set: ML_SET[G]
to_bag: ML_BAG[G]
ML_SEQ[G]
appended_by, infix “|>”: ML_SEQ[G]
{^ML_COLLECTION.extended_by}
prepended_by, infix “|<”: ML_SEQ[G]
remove (i: INTEGER): ML_SEQ[G]
item alias "[]" (i: INTEGER): G
domain: ML_SET[INTEGER]
head, last: G -- head = Current[0], tail = Current[count-1]
front, tail: ML_SEQ[G] -- tail is everything except `head`
override (i: INTEGER; x: G): ML_SEQ[G]
is_subseq_of, infix “|<<=|” (other: ML_SEQ[G]): BOOLEAN
ML_SET[G]
extended_by, infix “^” (x: G): ML_SET[G]
remove (x: G): ML_SET[G]
union, infix “+” (other: ML_SET[G]): ML_SET[G]
intersection, infix “*” (other: ML_SET[G]): ML_SET[G]
difference, infix “-” (other: ML_SET[G]): ML_SET[G]
is_subset_of, infix “|<<=|” (other: ML_SET[G]): BOOLEAN
is_disjoint_from, infix “|##|” (other: ML_SET[G]): BOOLEAN
override (x, y: G): ML_SET[G]
from_an_item (x: G): ML_SET[G]
ML_BAG[G]
ML_MAP[G, H]
has_key (k: G): BOOLEAN
extended_by_pair, infix “^” (p: ML_PAIR[G,H]): ML_MAP[G, H]
extended_by (k: G; v: H): ML_MAP[G, H]
remove (k: G): ML_MAP[G, H]
item alias "[]" (k: G): H
domain: ML_SET[G]
range_bag: ML_BAG[H]
union, infix “+” (other: ML_MAP[G, H]): ML_MAP[G, H]
intersection, infix “*” (other: ML_MAP[G, H]): ML_MAP[G, H]
difference, infix “-” (other: ML_MAP[G, H]): ML_MAP[G, H]
is_disjoint_from, infix “|##|” (other: ML_MAP[G, H]): BOOLEAN
override (x: G; y: H): ML_MAP[G, H]
from_two_arrays
(k: ESV_ARRAY[G]; v: ESV_ARRAY[H]): ML_MAP[G, H]
from_two_lists
(k: ESV_LIST[G]; v: ESV_LIST[H]): ML_MAP[G, H]
from_table (t: ESV_TABLE[G, H]): ML_MAP[G, H]
to_seq: ML_SEQ[ML_PAIR[G, H]]
to_set: ML_SET[ML_PAIR[G, H]]
to_bag: ML_BAG[ML_PAIR[G, H]]
ML_HASH_MAP[G, H->HASHABLE]
from_hash_table (t: HASH_TABLE[H, G]): like Current

Figure 2.3: Core Classes in the Mathematical model Library (ML)



(a) BON Diagram of STACK

```
class  STACK[G] feature

    put (x: G) is
      do
          imp.force (x, imp.count)
      ensure
          model |=| old model |> x
      end
      ...
end
```

(b) *put* feature of STACK

```
class  STACK_PROPERTIES[G] feature

    lifo (s: STACK[G] ; x: G) is
      require
          s /= void
      do
          s.put (x)
          s.remove
      ensure
          s.model |=| old s.model
      end
      ...
end
```

(c) Stack LIFO property

Figure 2.4: `STACK[G]` modeled by `ML_SEQ[G]`

items in the stack). The contracts of all the other features of the stack can be described in terms of the sequence and `count`. In the absence of a sequence to model the stack (i.e., with just the model attribute `count`), the best basic postcondition for the stack push operation `put` is:

$$count = \textbf{old}\ \ count + 1\ \ \textbf{and}\ \ item = x \tag{2.2}$$

However, this specification is incomplete. For example, an implementor can satisfy the above specification yet change old values of the stack that are not at the top. Therefore, we need a *frame condition* that says the old part of the stack remains unchanged. By adding a sequence to the model we can now express the complete contract as:

$$model \;\cong\; \textbf{old}\; model \;\blacktriangleright\; x \tag{2.3}$$

where, "▶" is the `appended_by` (pure) function of a mathematical sequence that returns a new sequence that is the same as the old one, but with the argument item appended to the end, "≅" means that left hand side and right hand side are *model equal* (as will be explained below). Since (2.3) → (2.2), there is then no need to write (2.2) as it is entailed by the model post-condition. With the full model, we can then provide the complete contracts for the pop operation `remove` and the query `item` that returns the top of the stack.

The Eiffel notation follows the BON notation quite closely as shown in Fig. 2.4b. For "▶", we may use either the `appended_by` function or alternatively the infix operator "`|>`".

Model classes such as `ML_SEQ` hold items that may be stored either by reference or by value (Eiffel has the expanded construct for constructing a value semantics). Given two mathematical sequences, $s_1, s_2 : ML\_SEQ[G]$, the assertion $s_1 \cong s_2$ (i.e., s_1 is *model equal* to s_2) holds precisely when the two sequences have the same number of elements and $s_1[i] = s_2[i]$ at each index i. The default interpretation of "$=$" is equality by reference. However, the meaning of the equality symbol can be changed to a value semantics if needed (in which case, the query `is_equal` in class `G` is used to compare values at index *i*). In Eiffel programs, we use the infix operator "`|=|`" for model equality. Model equality of the other collections are defined in the obvious way following this pattern.

With this specification of the stack, we may refine the specification to an efficient implementation. We may use efficient mutable structures such as a linked list or an array (e.g., `ARRAY` from the Eiffel base library).

Next, we need to define the abstraction relation [49] between the `model` query (which returns a `ML_SEQ`) and the concrete implementation `imp` which is an `ARRAY` (see Fig. 2.4). The abstraction function maps the concrete variables into the abstract objects which they represent. Thus, the body of the query `model` might be a loop that iterates through the implementation array and returns an equivalent sequence with the same elements as the array (i.e., we "lift" the mutable array into a mathematical immutable sequence). The postcondition of the abstraction function `model` is captured by the following postcondition:

$$Result \triangleq \langle i : INTEGER |\ 0 \leq i < imp.count \bullet imp\ [i] \rangle \tag{2.4}$$

where, the symbol "$\triangleq$" denotes equality by definition and the angle brackets $\langle\rangle$ stand for sequence comprehension (in the same way that $\{\}$ stands for set comprehension; e.g., $\{i : INT | 0 \leq i \leq 2i + 1\} = \{1, 2, 3\}$). Set, bag, sequence or map comprehensions are expressive notations supported by the ML library.

2.3 Scenario Tests as Specifications

Classical testing methodologies include unit, integration, system and acceptance tests [48]. Unit tests are usually thought of as tests that check a single method of a class. In conventional testing, unit tests are written at the end of code development.

```
test_characters_sorted: BOOLEAN is
    local
        sa: SORTABLE_ARRAY[CHARACTER]
    do
        sa := <<‘d’, ‘a’, ‘b’, ‘e’, ‘c’>>
        sa.sort -- use the sort routine
        check sa[1] = ‘a’ end
        check sa[2] = ‘b’ end
        check sa[3] = ‘c’ end
        check sa[4] = ‘d’ end
        check sa[5] = ‘e’ end
        Result := sa.count = 5
    end
```

Figure 2.5: A Unit Test (not a specification) for sortable array

Suppose, for example, a developer has already written a `sort` routine. A classic test for the `sort` routine is shown in Fig. 2.5. The test runs the `sort` routine on an unsorted array and then performs a sequence of concrete checks on the individual elements of the array to ensure that the correct element is inserted at

```
test_characters_sorted: BOOLEAN is
    local
        sa1, sa2: SORTABLE_ARRAY[CHARACTER]
    do
        sa1 := <<‘d’, ‘a’, ‘b’, ‘e’, ‘c’>>
        sa2 := <<‘a’, ‘b’, ‘c’, ‘d’, ‘e’>>
        sa1.sort
        Result := equal(sa1, sa2)
    end
```

Figure 2.6: A specification for sortable array

each index into the array.

Test Driven Development (TDD) [10] is a technique developed as part of the emerging Agile methodologies [3]. In TDD, a unit test is written before the `sort` method is implemented and is seen as a *test specification* of the method rather than just a sequence of checks as shown in Fig. 2.5. In the test specification, the sorted solution array is directly compared to the unsorted array (after application of the `sort` method). This captures the essence of what it means for the `sort` routine to succeed. First, the sorted array must be a permutation of the unsorted array, and secondly, the sorted array must be in increasing order. In TDD, the order is: (a) write a test; (b) develop enough code to satisfy the test; (c) refactor the code (if necessary) to improve the design while keeping the test specifications the same. The test in Fig. 2.6 is clearly closer to a specification that describes the *sortedness* property without constraining implementation details or algorithms.

This unit test is nevertheless limited, testing only the case of an array of characters for the specific case treated. It does not cover the case of an array of reals or some other class that conforms to a chosen total order, nor does it cover borderline cases such as where the array only has one element or is void. By contrast, the contracts written in the `SORTABLE_ARRAY` (Fig. 2.2) is a general specification that covers all these cases (any array in the constrained generic parameter `G`).

Tests have a narrower range than contracts in the sense that they only capture a specific set of scenarios in which the system will engage. However, they are concrete and fairly easy to write. The early and frequent nature of the tests (regression testing) helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious future debugging later in the project.

The classical unit test checks a small unit of code and not its interaction with other routines and modules. Likewise, contracts are good for specifying the obligations and benefits of clients with respect to the features of a single class.

However, specifications of systems are often use cases that involve collaborations between different modules in the system. Berry *et. al.* write as follows:

> An RS [requirement specification] is often accompanied by or includes descriptions of scenarios and use cases. These should be the same scenarios and use cases that describe how users exercise the CBS [computer based system] to do their work. These scenarios and use cases in turn form a good basis for building test cases that cover the expected ways the CBS will be used. Moreover, since the users are

> guided by the user's manual in their uses of the CBS, these test cases provide a good coverage of the expected uses of the CBS. [14]

In this thesis, we call such use cases and collaborations *Scenario Tests* and we use the standard Eiffel unit testing framework to write them. Instead of testing a single feature, a Scenario Test (by contrast) specifies a collaboration to achieve some functionality among various modules of the system.

Consider, for example, the Scenario Test (shown in Fig. 2.7) for a chat application[4]. This specifies a collaboration among various classes of the chat application such as `CHAT_SERVER`, `CHAT_ROOM` and `CHAT_USER`. The test specifies specific features in `CHAT_SERVER` such as: `users`, `rooms`, and `add_room (a_user:CHAT_USER)` by describing the following scenario:

1. Create a chat server (line 10)
2. Check that the newly created server has only one user (line 13) and only one chat room (line 14)
3. Create two chat users ("Mike" and "Anna") and connect them to the server (lines 16–19)
4. Check that the new users are connected and reside in the lobby room of the chat server (lines 20–22)

[4] The extended version of this example is described in Chapter 4.

5. A user (“Mike”) adds a room to the server named “Technical Support” (lines 24–25)

6. Check that the new room is successfully added to the server (line 26) and its status is public (line 27)

7. The owner of (“Technical Support”) room (“Mike”) changes the room status to private (line 30)

8. Check that the status of the room is changed to private (line 31)

9. The owner allows a user to join the room (“Mike” allows “Anna” to join “Technical Support”)

10. Check that “Anna” is allowed to enter “Technical Support” (line 35)

The benefit of this type of test is that it helps to derive the design, e.g., if all the classes and feature signatures of the above system (that are referred to in the Scenario Test) are added to the system (such that project compiles), the design illustrated in the BON class diagram in Fig. 2.8 will be automatically generated. The class diagram presents a design of the system (classes and feature signatures, but not yet code in the bodies of the features).

The Scenario Test will fail if: (a) the collaboration between the various elements fails to satisfy the specified checks or to produce the anticipated results or

```
scenario_test: BOOLEAN is
    local
       server: CHAT_SERVER
       mike, anna: CHAT_USER
       mike_room: CHAT_ROOM
       users: LIST[CHAT_USER]
       rooms: LIST[CHAT_ROOM]
    do
       -- create the chat server and check it
       create server.make
       users := server.users
       rooms := server.rooms
       check server.user_count = 1 end
       check server.room_count = 1 end
       -- create 2 users Mike and Anna and connect them to the server
       create mike.make ("Mike")
       create anna.make ("Anna")
       server.connect (mike)
       server.connect (anna)
       check server.user_count = 3 and server.room_count = 1 end
       check mike.room = server.lobby and anna.room = server.lobby end
       check users.has(mike) and users.has(anna) end
       -- Mike creates and adds a room ‘‘Technical Support"
       mike_room := mike.create_room ("Technical Support")
       mike.add_room (mike_room)
       check server.room_count = 2 end
       check not mike_room.is_private end
       check rooms.has(mike_room) end
       -- Mike changes the status of his room to private
       mike.set_private ("Technical Support")
       check mike_room.is_private end
       check not server.is_allowed (anna, "Technical Support") end
       -- Mike allows Anna to join the Technical Support room
       mike.allow_user ("Anna", "Technical Support")
       check server.is_allowed (anna, "Technical Support") end
       Result := True
    end
```

Figure 2.7: Scenario Test

(b) the contracts fail while executing the tests. Scenario Tests (as in Fig. 2.7) thus do two things for us. Firstly, they specify the design. Secondly, they specify such

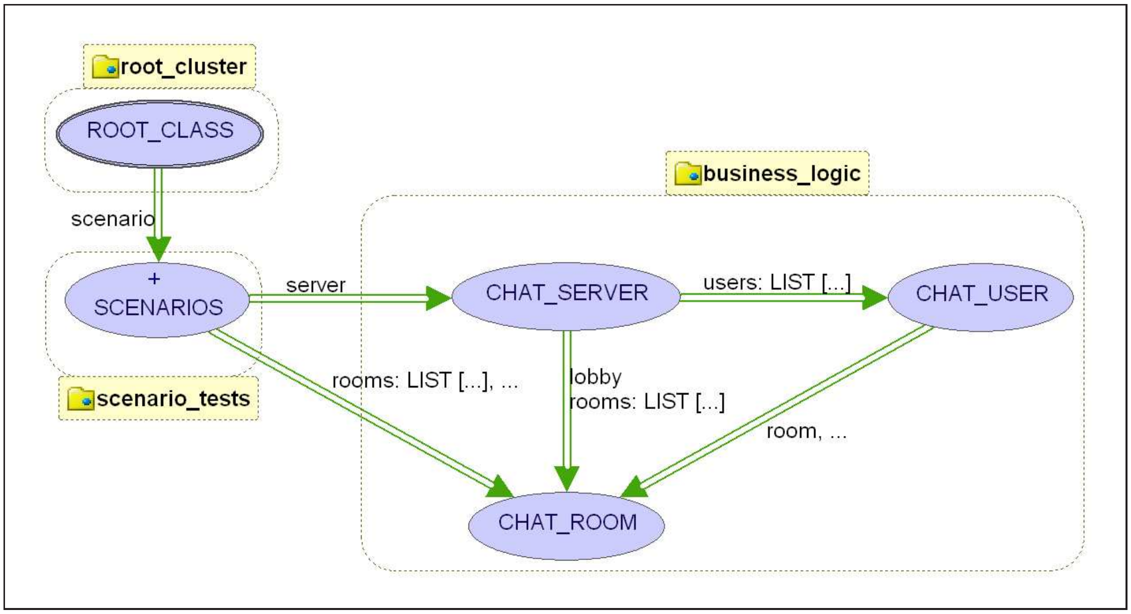


Figure 2.8: Design consequence of the Scenario Test

design in a mechanically testable manner.

2.4 Synergy between ML-Contracts and Scenario Tests

ML-Contracts provide precise general specifications covering all states of the program, and Scenario Tests execute and check the correctness of the ML-Contracts with respect to the specific data used. There is thus a synergy between the ML-Contracts and the Scenario Tests. They both specify aspects of the design and both are mechanically checkable. ML-Contracts act as test amplifiers, i.e., when we execute the tests, all ML-Contracts will also be executed and tested. Together they provide precise *Testable Specifications* of the future software product. By writing such testable specifications early we:

1. Communicate with clarity to all developers what the future product must do.
2. Provide a concrete testable criterion to indicate when this job is completed.
3. Provide precise readable documentation guaranteed to be up to date with the code.
4. If we work incrementally (as in TDD), we can add new units of functionality while at the same time providing regression testing of the already implemented functionality.
5. Provide the developer with a safety net to refactor (change the implementation without changing the functionality), as the testable specifications can be re-run after any changes to check that the functionality is preserved.

2.5 ES-Test for Testable Specifications

ES-Test (see Fig. 2.1) is the part of the ESpec tool that handles testable specifications (ML-Contracts and Scenario Tests). ES-Test has a unit testing framework that provides facilities for developers to write and execute unit tests similar to JUnit [41] for Java. This facility can be used to write Scenario Tests as well.

However, ES-Test also supports contracts. Thus, contract violations are reported in the GUI and accurately pinpoint the failing contract clauses of a given feature (tests may be run from the command line as well). Two kinds of test cases are supported: Boolean Tests and Violation Tests.

ES-Test is a successor to an Eiffel unit testing library called E-Tester reported in [73]. The contribution of this author to further development of the tool includes:

1. Allowing unit tests to be integrated with other checks (such as Fit tests and static verification) in a single test suite.

2. The addition of Tagged Violation Tests (see section 2.5.2).

3. Complete reports of the type of contract violation and the tag involved in the violation (both in the GUI and the command line version of the tool). A developer may use the `show_error` command to reflect the complete call stack in the report (see Chapter 5).

2.5.1 Boolean Test

ES-Test supports *Boolean Test* case and *Violation Test* case. A Boolean Test is a query routine that returns a `BOOLEAN` result. This type of test checks the System Under Test (SUT) by invoking program features, and then checking that the state of the computation satisfies certain conditions. The test passes if and only if the conditions are true and all the contracts are satisfied (i.e., there are no contract violations or runtime exceptions). If a Boolean Test case succeeds, i.e., it terminates with a true result, then all the contracts invoked during the test have also succeeded and we have thus partially checked the correctness of the SUT.

If any contract is violated during the execution of the test, the violation will be reflected in the final report of the ES-Test tool. This report also provides additional debugging information for the developer to fix the problem. A precondition violation during the execution of a test tells us that the calling class is at fault. Any postcondition, invariant or check instruction violation tells us that the supplier is at fault.

An example of a Boolean Test case was already shown in Figures 2.7 and 2.6. For detailed explanation on how to execute such tests, please see the ESpec's user manual in Appendix D.

2.5.2 Violation Test

If a Boolean Test succeeds, then there are no contract violations for the given scenario. By contrast, a *Violation Test* calls a routine in a state in which it is expected that the precondition (or the class invariant) is violated and then checks that the violation occurs. Such a test succeeds only if the expected contract violation occurs.

Why would we want to test for violations? A Boolean Test can never check that a feature has an appropriate precondition (or class invariant), whereas a Violation Test can check for a missing precondition by calling the feature in a state that violates the precondition.

At times, the precondition exists, but is only partially correct. For example, suppose the precondition for a routine is $i > 0$ (accept only positive values for *i*), and suppose instead, that the developer writes the incorrect precondition $i \geq 0$. A Violation Test might call the routine with $i = 0$ expecting a contract violation. When the expected contract violation fails to occur, the developer is informed that the precondition is incompletely specified.

As another example, consider the `put` feature of class `DICTIONARY` (shown in Fig. 2.9) which inserts a *key* and its associated *value* into the dictionary. The precondition of this feature (captured as an ML-Contract) asserts that the new key should not already be a member of the dictionary ($key \notin model$), where the `model` is a mapping between the inserted *keys* and the corresponding *values*. The construction of this model is done at lines 25–36 of Fig. 2.9 using the `ML_MAP` class of the ML library.

A Violation Test is a command routine (as opposed to a Boolean Test which is a query). It is expected that the command routine will generate an exception

```
class
  DICTIONARY [KEY, VALUE]
...
  put(v: VALUE; k: KEY) is
      -- Put key ’k’ into the dictionary with associated value ’v’
    require
      key_not_in: not model.has_key (k)
    local
      dictionary_item: DICTIONARY_ITEM[VALUE, KEY]
    do
      create dictionary_item.make(v, k) -- create an item with key ’k’ and
          value ’v’
      container.extend (dictionary_item)
    ensure
      count_incremented: count = old count + 1
      new_key_value_added: model |=| ((old model).extended_by (k, v))
      check_value: equal((model.item (k)), v)
    end

feature {NONE} -- implementation

  keys: LINKED_LIST [KEY]
  values: LINKED_LIST [VALUE]

feature -- model
  model: ML_MAP [KEY, VALUE] is
      -- what is the implementation of the current dictionary?
    local
      set_pairs: ML_SET [ML_PAIR [KEY, VALUE]]
      a_pair: ML_PAIR [KEY, VALUE]
    do
      create set_pairs.make
      create a_pair
      set_pairs := a_pair.from_parallel_lists (keys, values)
      create Result.make_from_pair_set (set_pairs)
      Result.compare_objects
    end
...
```

Figure 2.9: Put feature of a dictionary

during its execution. An example of a Violation Test for the dictionary is shown in Fig. 2.10 which invokes the put routine with a key which is already in the dictionary. This test case, creates a DICTIONARY object (line 11) and then inserts the same key “key1” twice (lines 12 and 13) expecting that an exception should be generated, thus checking that the precondition is correctly and completely expressed.

```
 1  put_violation is
 2    local
 3      a_dictionary: DICTIONARY[INTEGER, STRING]
 4      v: INTEGER
 5      k1, k2: STRING
 6    do
 7      comment("Fails to put a key which already exists in the dictionary")
 8      v := 123
 9      k1 := "key1"
10      k2 := "key1"
11      create a_dictionary.make
12      a_dictionary.put(v, k1)
13      a_dictionary.put(v, k2) -- should fail here
14    end
```

Figure 2.10: A Violation Test Case for put feature of the DICTIONARY

ESpec provides another type of violation test called a *Tagged Violation Test*. This kind is similar to the standard violation case; however, a specific contract violation is expected.

Contracts are written as a sequences of clauses. Each clause may optionally have an associated description tag (e.g., the tag key_not_in in Fig. 2.11). A Tagged Violation Test provides a specific tag as input and succeeds only if there is a contract violation associated with the clause associated with that tag.

2.5.3 Collections of Test Cases

ES-Test provides the ability to group a set of test cases (either Boolean or Violation tests) together in a single class (that inherits from `ES_TEST`). A test suite collects together such groups of tests. A test suite is constructed in a class that inherits from `ES_TEST_SUITE` (see Chapter 5 for more information).

An example of a group of test cases is provided in class `DICTIONARY_TEST` (shown in Fig. 2.11) which is a descendant of `ES_TEST` class. A test may be added to the group (in the `make` routine) either as a Boolean, Violation or Tagged Violation case (lines 9–16 in Fig. 2.11). The command `add_violation_case` (which is inherited from `ES_TEST` class) is used to add a standard violation test.

To add a Tagged Violation case, we use the `add_violation_case_with_tag` (also inherited from `ES_TEST`) and we provide the exact name of the expected violation tag. Similarly, the `add_boolean_case` command is used to add Boolean test cases.

The `run_espec` command (line 15 in Fig. 2.11) executes all the test cases specified in the creation routine of class (`DICTIONARY_TEST`).

The results will then be collected and reported to ESpec's GUI. Any type of exception such as precondition, postcondition, invariant, check instruction, loop variant, and loop invariant violations as well as other errors (e.g., an OS error)

```
class
  DICTIONARY_TEST
inherit
  ES_TEST -- All test unit classes inherit from ES_TEST
create
  make
feature

make is
    -- In the make routine we define test cases associated with this class
  do
    add_boolean_case (agent dictionary_test_put)
    add_violation_case (agent put_violation)
    add_violation_case_with_tag ("key_not_in", agent put_violation)
    run_espec
  end

feature -- Test Cases (Boolean/Violation) are written in this section
  ...
end
```

Figure 2.11: Test Unit class that contains number of test cases

will be reported directly to the GUI. The test cases can be run individually or in unison with feedback to a single green/red bar. A snapshot of ESpec GUI was shown in Fig. 2.1.

2.6 Conclusion

We have argued in this chapter that it is the automatic testability of specifications that is important. Testable specifications are very concrete—they either succeed or they fail, and then you know if the two viewpoints (specifications and implementations) are consistent.

In the first chapter we described the development process as follows:

1. Specification correctness: $P \wedge S \rightarrow R$
2. **Implementation correctness:** $\boldsymbol{C \rightarrow S}$
3. System correctness: From (1) and (2) conclude that: $P \wedge C \rightarrow R$

In this chapter we have shown how ESpec can be used to test specifications—formula (2). The next chapter discusses how we can test requirements—formula (1)—in such a way that testable requirements and testable specifications can be checked together in an integrated fashion.

3 Testable Requirements

The primary measure of success of a software system is the degree to which it meets the purpose for which it was intended. Requirements engineering is the process of discovering that purpose, by identifying stakeholders and their needs, and documenting these in a form that is amenable to analysis, communication, and subsequent implementation [69].

A requirement will generally avoid saying *how* the system should be implemented leaving such decisions to the designer. According to [89], a high quality requirements document must be:

- Correct—Each requirement must accurately describe the functionality to be delivered. The customer is the ultimate authority to determine the correctness of the requirement.

- Unambiguous—The reader of a requirement statement should be able to draw only one interpretation of it.

- Complete—No necessary information should be missing from the requirements.

- Consistent—A requirement should not conflict with other requirements or with higher level business rules. Disagreements among requirements must be resolved before development can proceed.

- Verifiable—The customer must be able to determine if the requirements have been met. If a requirement is not verifiable, determining whether it was correctly implemented becomes a matter of subjective opinion. Requirements that are ambiguous are thus not verifiable.

A *Testable Requirement* is a consistent, unambiguous description of the expected system behaviour that is verifiable. The question is how do we make requirements verifiable?

In the first chapter, we distinguished between a design specification S and a customer requirement R in a problem domain described by P. As asserted in Formula (1) in Section 1.3, the relationship between the customer's requirement and a design specification is $P \wedge S \rightarrow R$ (i.e., any behaviours that satisfy the specification also satisfy the requirements). To test for the presence of requirement R is to check that the suggested design solution S entails the requirement.

If the requirements are described informally (e.g., as English text) then there is no real way to test them mechanically. Furthermore, as the solution S is refined, we will want to check $P \wedge S \rightarrow R$ repeatedly (as in continuous regression testing). So it would be advantageous to mechanize requirement testing. This means that requirements must be formalized in order to mechanize requirement testing.

Our approach will be to use the Fit framework [68] to formalize requirements in a language understandable to customers (i.e., it is not a programming language). The benefit of the Fit framework is that business people and customers can define and edit the tests according to what they believe is the correct function of the code.

3.1 Fit Framework

The Fit framework (Fit) [68], was developed by Ward Cunningham as an attempt to fill the gap between developers and the customers. Customers cannot be expected to write Scenario Tests or ML-Contracts as these require special programming skills. What the Fit framework does is it allows customers with no programming background to write acceptance tests for software products in the form of understandable tables in their word processor or spreadsheet application (provided that HTML can be generated by the application). Customers can use freely available web browsers or HTML editors to view and edit their tests.

Incorrect or incomplete requirements is a problem in software projects and it is usually due to lack of understanding on the part of developer of the customer requirements. According to [45]:

> Lack of user involvement traditionally has been the No. 1 reason for project failure. Conversely, it has been the leading contributor to project success. Even when delivered on time and on budget, a project can fail if it doesn't meet user needs or expectations...

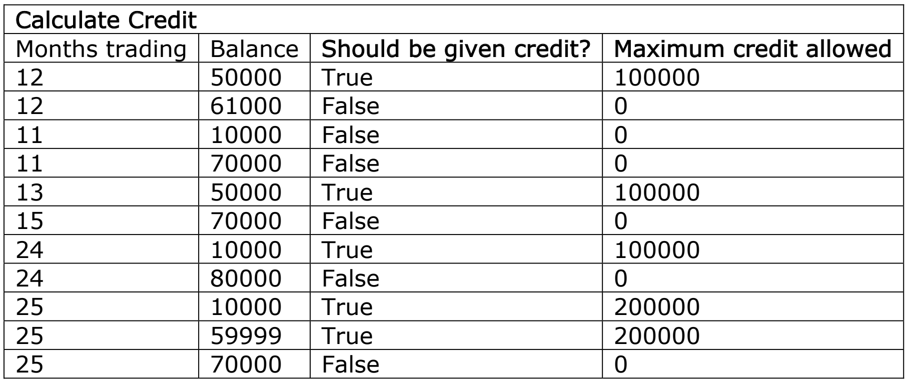
Fit enhances the collaboration in the development process by allowing customers to lend their experience in the subject matter to the effort. Fit automatically compares customer's expectations to actual results of testing and gives them a way to see what software really does.

An important benefit of Fit is that it encourages thinking about the problem domain (as opposed to the solution space), in the same way that Test Driven Development encourages thinking about design (as opposed to implementations).

As an example, consider the following business rule that describes the calculation of customer credit limits:

> **[R1]** "Credit is allowed, up to an amount of $100,000 for companies who have been trading with us for at least one year and have a balance owing of less than $60,000. This credit is extended to an amount of $200,000 for companies who have been with us for more than two years."

Our banking customer may use an editor to create the Fit table shown in Table 3.1 which is an example of a "Column Table" (which will be discussed in more detail in the following sections). This table is a concrete and testable description of the business rule [R1].



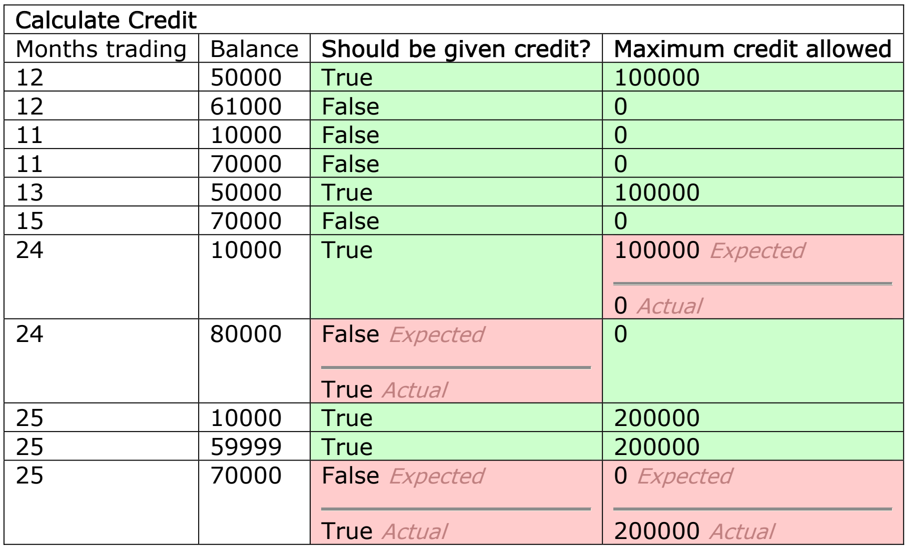
Calculate Credit			
Months trading	Balance	Should be given credit?	Maximum credit allowed
12	50000	True	100000
12	61000	False	0
11	10000	False	0
11	70000	False	0
13	50000	True	100000
15	70000	False	0
24	10000	True	100000
24	80000	False	0
25	10000	True	200000
25	59999	True	200000
25	70000	False	0

Table 3.1: Fit Table describing a set of concrete examples related to [R1]

The first row of the table contains the table name. The second row contains the column headings. There are two types of column headings: The first two (*Months trading* and *Balance*) are the given inputs and the next two (*Should be given credit?* and *Maximum credit allowed*) are the expected values for testing the creditworthiness of a company. The client is able to check the table against the code that developers have been working on, and see the output shown in Table 3.2 (passed cases are colored green[5]).

Each row is an independent test case. For example, the third row illustrates a passed test, i.e., a company has been trading for one year and has a balance less than $60,000. The 9th and 10th rows show failed tests, i.e., the system under test generated outputs which were not expected for the case. Inspecting these two failed cases shows that the implemented code has not correctly calculated the

[5] A color mapping chart is provided in the Appendix E (see Table E.1) for understanding the black and white copies of this thesis.



Calculate Credit			
Months trading	Balance	Should be given credit?	Maximum credit allowed
12	50000	True	100000
12	61000	False	0
11	10000	False	0
11	70000	False	0
13	50000	True	100000
15	70000	False	0
24	10000	True	100000 *Expected* 0 *Actual*
24	80000	False *Expected* True *Actual*	0
25	10000	True	200000
25	59999	True	200000
25	70000	False *Expected* True *Actual*	0 *Expected* 200000 *Actual*

Table 3.2: Result of running table 3.1

output values for the boundary case of 24 months. As shown in Table 3.2, these errors are directly reported to the Fit table with both the *actual* value (as returned by the system) and the *expected* value (asserted by the customer) highlighted.

The last row of the table shows another failed case. This case expects that implemented code disallows credit for a company which has been trading for *more* than two years (25 months) and has a balance of $70,000. Comparing this test case to the informal requirement [R1], we can see that [R1] does not clearly specify the required balance for giving credit to a company who has been trading for more than two years (i.e., [R1] says "...This credit is extended to an amount of $200,000 for companies who have been with us for more than two years" but never clearly say anything about the balance of those companies). However, the tables clearly expects that *no credit* should be given to such company if its balance is $70,000.

The failed cases in the Fit table usually generate discussions between the developers and the customers. These discussions are valuable because they give the developers a better understanding of what the customers really want. After further investigations about this particular case, developers understood that the required balance for giving credit to a company remains the same (less than $60,000) even for companies who have been trading for more than two years. The informal requirement [R1] assumed that it was understood from the context that the balance requirement is the same as before (less than $60,000); however, an upfront concrete test like Table 3.2 from the client revealed that the system under test did not achieve such requirement (see Section E.1 for the Fixture source code).

Fit creates a feedback loop between customers and programmers. It's an invaluable way to collaborate on complicated problems—and get them right—*early* in development. Informal requirements alone are inadequate (as we saw in case of [R1]), especially when they have to be completed without feedback from the development process. Having concrete tests that are based on realistic examples from the business domain help build a common understanding of the business needs.

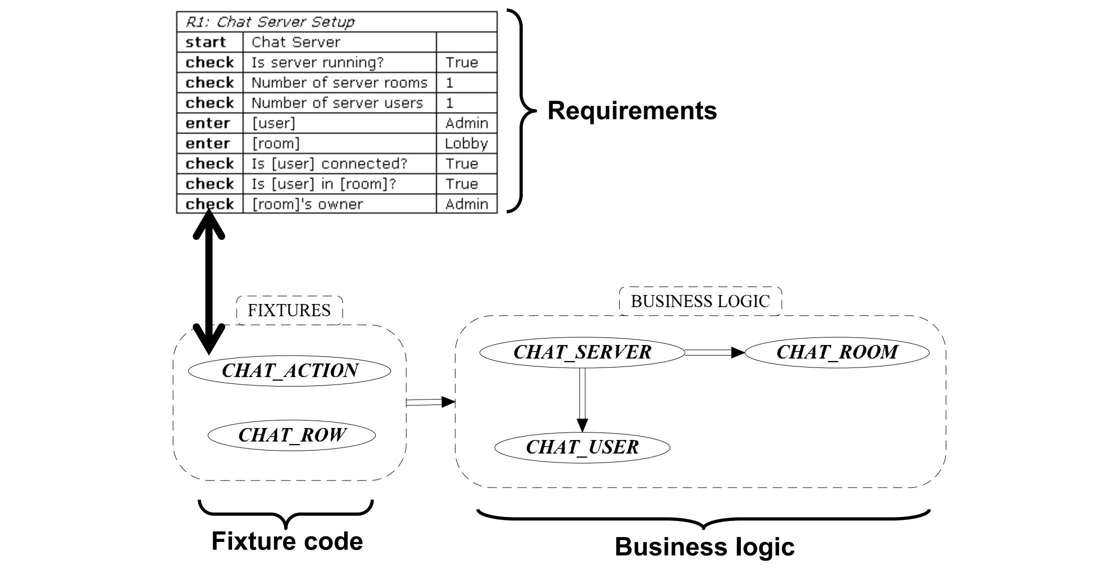
We can summarize the benefits of Fit tables as follows [68]:

- **Communication:** providing a way for people who want a system to discuss and communicate that need in a concrete way.
- **Agility:** keeping the software in good shape by supporting design changes that are essential as the needs of the business change. Automated tests help define those changes and help ensure that any changes to the software do not break previously satisfied requirements.
- **Balance:** spending less time on gaining balance with fixing problems by reducing the number and severity of problems, catching them early, and making sure they don't return.

3.2 ESpec support for Fit

ESpec's Fit engine (ES-Fit) is the first implementation of the Fit Framework for the Eiffel language. ES-Fit supports all the official tables as described in the original Fit framework. Developers and/or the customers can run these tests against the implementation under development to see if it behaves correctly.

How does the developer satisfy the requirements specified in the customer-provided Fit table? The developer will need to write two kinds of classes: *Fixture* classes and classes of the business logic (see Fig. 3.1). Fixtures are glue code between the customer-provided requirements and the business logic. ES-Fit provides libraries that allow the developer to easily develop such Fixtures that connect the requirements to the business logic. ES-Fit uses the developer written Fixture classes to parse the requirement document, extract the tables, interpret the tables and invoke the relevant business logic and then reflect the results of running the business logic back to the tables in the requirements document. The rows in tables where the checks succeed are coloured green and those that fail are coloured red.



R1: Chat Server Setup		
start	Chat Server	
check	Is server running?	True
check	Number of server rooms	1
check	Number of server users	1
enter	[user]	Admin
enter	[room]	Lobby
check	Is [user] connected?	True
check	Is [user] in [room]?	True
check	[room]'s owner	Admin

Figure 3.1: Relationship between Fit tables, Fixtures and the Business logic.

Fixtures define how the underlying Fit engine should read and execute each table of the HTML document. ES-Fit implements three types of Fixture classes that correspond to each of the table types in the original framework, namely: `ES_COLUMN_FIXTURE`, `ES_ACTION_FIXTURE`, and `ES_ROW_FIXTURE`. ESpec adds number of extensions to the original Fit framework:

- **Introducing Design by Contract (DbC) into the Fit Framework:** Contract violations will be reported directly to the tables allowing the customers to give valuable feedback to the developers right from the start of the development process. This helps developers catch bugs in the specifications.
- **Flexible method for constructing new Fixture types:** ES-Fit allows the developer to redefine—and therefore change the behaviour of—the default Fixtures in order to create new desired types of Fixtures (implementation detail will be discussed in Chapter 5).
- **Addition of Reference Tables:** The ability of a Fixture to reference common data that is shared between Fit tables simplifies the description of the requirements (see Chapter 5).
- **Flexible naming conventions:** ES-Fit does not force customers to follow a strict naming convention. Customers can directly use business terms as they appear in the problem domain (i.e., customers don't need to know anything about the solution domain such as Fixtures classes, business logic classes, etc.). This is an improvement to the original Java implementation

of the Fit framework in which customers are restricted to use particular names in their tables (e.g., in the Java version, header names of the table should be the same as the Fixture class or header name of each column should be exactly defined as the function name defined in the Fixture code).

ESpec works in either the command line mode or the GUI mode (see Appendix D). The developer and or the customer can select the input HTML file (or a directory in case there are number of HTML inputs) and then press "Run ES-Fit" button on ESpec's GUI. This will invoke ES-Fit. ES-Fit will read the input HTML file and execute the Fixture code associated with tables defined inside the HTML document and reports the results back to the tables. If any of the Fit tests fail, a red bar will be displayed to the user. The user can then click on the failed Fit tests to see the failures.

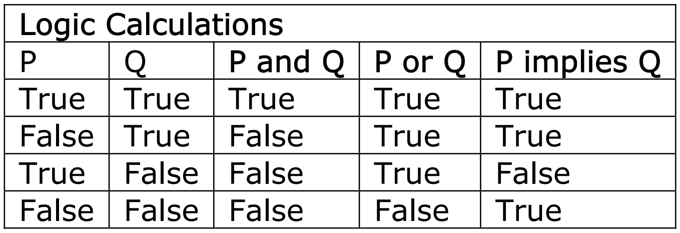
After Fit tables are provided by the customers, the next step for the developers would be to implement the Fixture code. Fixtures drive the development process by forcing the developers to write enough code (e.g., classes and feature declarations) so that the system becomes compilable.

In the following sections, we introduce different types of default Fixtures which are supported by ES-Fit. For more information regarding the implementation and design of the ES-Fit tool, please see Chapter 5.

3.2.1 Column Fixture

The ES_COLUMN_FIXTURE class, provides facilities for the developers to define a Fit Column Fixture. A Column Fixture is used to test the calculations or decisions that are made by the system under test. A Column Table captures the business logic (in the problem domain) by allowing the customers to define business logic in a tabular format with number of columns corresponding to the input data and a few more columns for expected results. A sample Column Table was already shown in Table 3.1.

Table 3.3 is a simpler example of a Column Table. The header is the name of the table and is a string arbitrary chosen by the customer. The first two columns of this table (*P* and *Q*) are the input Boolean values and the next three columns (*P and Q*, *P or Q* and *P implies Q*) are the expected Boolean outputs.



Logic Calculations				
P	Q	P and Q	P or Q	P implies Q
True	True	True	True	True
False	True	False	True	True
True	False	False	True	False
False	False	False	False	True

Table 3.3: A sample Column Table

Fit ignores any formatting applied to table cells; italicized, bold, or underlined text can be used to highlight important rows or values in the table without affecting Fit's ability to execute the test. Fit executes tests one row at a time, from left to right of the column order.

The Fixture code associated with this table is shown in Fig. 3.2. The Fixture class is a concrete subclass of `ES_COLUMN_FIXTURE` (line 4). The creation routine `make` (lines 10–15) binds the customer-provided names of the calculations (in the Fit table) to the appropriate agent function defined in the body of the Fixture class. For example, string "P and Q" is bound to `calculate_and` agent at line 12.

For simple examples, the business logic resides in the Fixture code. For example, in Fig. 3.2 the calculation of *"P implies Q"* given by `Result := a implies b` is contained in the Fixture code (lines 29–32). Obviously, as the code increases in complexity the developer will want to develop design classes. The job of the Fixture code will be to call the appropriate features of the business logic.

In order to run the Fit tests, the developer needs to define a root class (shown in Fig. 3.3) in which, the name of the tables are bound to the corresponding Fixture classes, e.g., every table with name *"Logic Calculations"* is bound to the `LOGIC_FIXTURE` class.

ES-Fit can be invoked either through the command-line or through the ESpec GUI. ES-Fit reads the input HTML file (or a directory containing HTML files) and runs the Fit tables against the system under test. Fig. 3.4 shows the result of executing the "Logic Calculation" table.

```
class
  LOGIC_FIXTURE
inherit
  ES_COLUMN_FIXTURE

create make

feature -- creation

  make -- binding is done in here
    do
      bind ("P and Q", agent calculate_and)
      bind ("P or Q", agent calculate_or)
      bind ("P implies Q", agent calculate_implies)
    end

feature -- agents for calculations

  calculate_and (a, b: BOOLEAN): BOOLEAN
    do
      Result := a and b
    end

  calculate_or (a, b: BOOLEAN): BOOLEAN
    do
      Result := a or b
    end

  calculate_implies (a, b: BOOLEAN): BOOLEAN
    do
      Result := a implies b
    end
end -- class LOGIC_FIXTURE
```

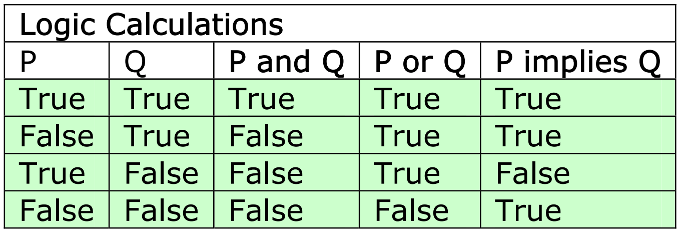
Figure 3.2: The Fixture code associated with the Column Table 3.3

When a table is checked, the cells representing expected outcomes are shaded green, red, yellow, or gray; green means that expected and actual values match, red means they didn't match (in which case expected and actual appear in the cell), yellow indicates that an unexpected error happened or a contract violation

```
class -- Root class of the system
  ROOT_CLASS
inherit
  ES_SUITE
create
  make

feature -- binding the name of the table to the associated Fixture class
  make
    do
      add_fixture ("Logic Calculations", create {LOGIC_FIXTURE}.make)
      run_espec
    end
end -- class ROOT_CLASS
```

Figure 3.3: System ROOT_CLASS for running Table 3.3

was thrown (a stack trace appears in the cell directing the developers to the location of the problem), and gray means that the field or method is not implemented in the Fixture class or that the cell was ignored by ES-Fit.



Logic Calculations				
P	Q	P and Q	P or Q	P implies Q
True	True	True	True	True
False	True	False	True	True
True	False	False	True	False
False	False	False	False	True

Table 3.4: Result of executing Table 3.3

3.2.2 Action Fixture

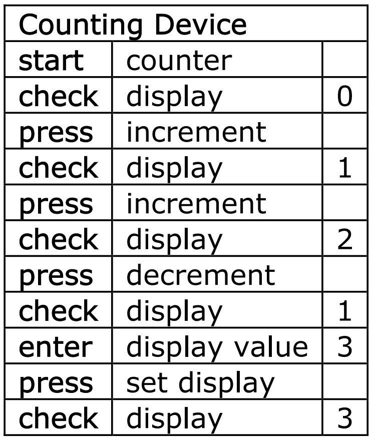
Any class that is a descendant of ES_ACTION_FIXTURE class, becomes an Action Fixture. An Action Fixture tests that a series of actions carried out on an application works as expected. An Action Fixture starts a class from the underlying system by creating an instance of that class. Subsequent actions are made through feature calls on that object.

An Action Table (input to an Action Fixture) is created by the customers to define a sequence of actions to be executed on the underlying system. Customers can express the expected behaviour of the system when such sequence of actions are carried out. Each row in an Action Table defines a single action. Actions are defined with the help of following keywords (each row must start with a keyword):

- **start** *app*: create/reset the application *app*.
- **enter** *act arg*: run action *act*, on the application *app*, and provide an input argument *arg*.
- **press** *act*: run action *act*, on the application *app* (no argument is provided).
- **check** *prop val*: check that the property *prop* has value *val* in the application *app*.

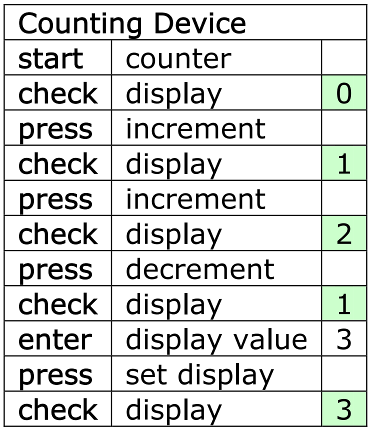
The **start** keyword in an Action Table causes the corresponding Action Fixture to create or re-initialize an object. It is the job of the developer to make the appropriate connections in the Fixture class (i.e., to decide which objects need to be created).

The **press** and the **enter** keywords execute commands that correspond to the actions defined by the customers. The **check** keyword runs a function query on the object which was originally initialized by the **start**.



Counting Device		
start	counter	
check	display	0
press	increment	
check	display	1
press	increment	
check	display	2
press	decrement	
check	display	1
enter	display value	3
press	set display	
check	display	3

Table 3.5: An Action Table for a counter device



Counting Device		
start	counter	
check	display	0
press	increment	
check	display	1
press	increment	
check	display	2
press	decrement	
check	display	1
enter	display value	3
press	set display	
check	display	3

Table 3.6: Result of Table 3.5

An Action Table for controlling and testing a counter device is shown in Table 3.5. The corresponding Fixture code is shown in Fig. 3.4.

In the first row of Action Table 3.5 the Customer provides an arbitrary title such as: “Counting Device”. In the first column of the table we can see keywords (**start**, **check**, **enter** and **press**) which we described above.

The keyword **start** is used to initiate the counter device. Usually, there is only one **start** per Action table. Thus, the second row of the table starts the business logic for the counter. If there is another Action Table in the same HTML document, it will use the current counter unless there is another **start** in that table (which would re-initialize the counter business logic).

The keyword used in the third row is **check**. It tests that a property (designated by the descriptive text in the second column) satisfies some value (specified by the text in the third column). The action in the third row thus states that the counter "display" must have the value "0".

Properties of the business logic are specified in the second column of the Action table. The customer may use any descriptive string (say *Str*) to denote a property (say *Prop*) in the second column. Once *Str* is specified then it always denotes the same property *Prop* throughout this table and any other Action table. Values in the third column of the Action table are interpreted by the Fit framework as booleans, integers, reals, characters, strings and arrays of the basic types. As far as the customer is concerned, a value is just a descriptive string (e.g., "0", "1", "2", etc...).

The keyword **press** in Table 3.5 denotes an action that effects some change in the business logic (incrementing the value of the counter). The keyword **press** may be used together with **enter** to denote a parameterized action, e.g., we may use "**enter** *display value*" together with the action "**press** *set display*" to change the value of the counter to an arbitrary value.

The **start** keyword causes the Fixture code to run the `start` routine (lines 22–23 in Fig. 3.4) by passing the value of the second cell ("counter") as the argument.

```
class COUNTER_FIXTURE inherit
  ES_ACTION_FIXTURE
create
  make

feature{NONE}
  make is -- Binding is done in here
    do
      bind("increment", agent increase)
      bind("decrement", agent decrease)
      bind("display", agent display)
      bind("display value", agent set_display_value)
      bind("set display", agent set_display)
    end

  counter: COUNTER -- Global object created by start

  display_value: INTEGER

feature -- Actions to be invoked on the system under test

  start (arg: STRING) is
    do create counter.make end

  display: INTEGER is
    do Result := counter.display end

  increase is
    do counter.increase end

  decrease is
    do counter.decrease end

  set_display is
    do counter.set_counter (display_value) end

  set_display_value (v: INTEGER) is
    do display_value := v end

end -- class COUNTER_FIXTURE
```

Figure 3.4: The Fixture code associated with the Action Table 3.5

This argument value could be used to create variety of applications in the `start` routine.

The **check** keyword executes the `display` (lines 25–26) routine and then compares the returned value "0" to the expected value in the table. If there is a match, then the corresponding cell will be marked with green. Binding the names appeared in the table to the associated agent routines is done in the creation routine `make` (lines 7–14) this is similar to a Column Fixture.

The **enter** action calls the routine `set_display_value` (lines 37–38) which is bound to the name in the second column ("display value") passing "3" as its argument.

Another way to see an Action Table is to think of it as an "imaginary Graphical User Interface (GUI)" with various empty text fields and buttons (similar to Fig. 3.5). The job of an Action Table, is to mimic an imaginary customer who enters text into the text fields of a GUI and then clicks on various GUI buttons in order to test the underlying application.

3.2.3 Row Fixture

A Row Table tests whether the expected elements of a collection (or database) matches the actual elements in the collection (or database).

In ES-Fit, the developer can define a Row Fixture class by inheriting from the

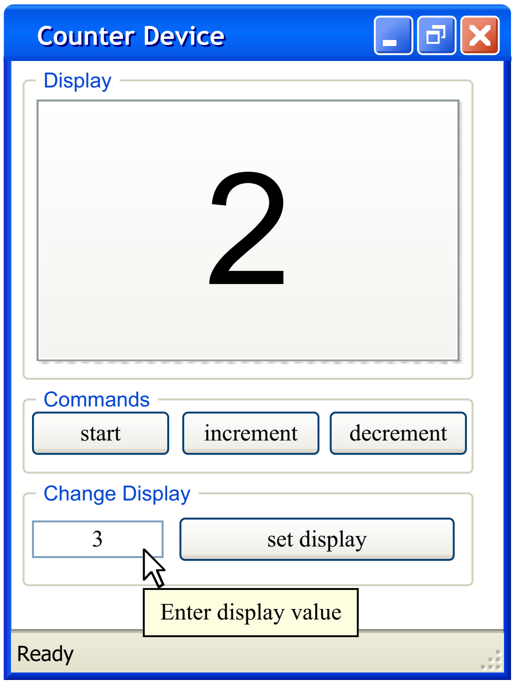


Figure 3.5: Action Fixture Table simulates an imaginary GUI

generic class ES_ROW_FIXTURE. An algorithm matches rows with objects based on one or more keys. Objects may be missing or in surplus and are so noted [68]. A simple example of a Row Table is shown in Table 3.7 which checks the contents of a phone book. The Fixture code associated with this example is shown in Fig. 3.6.

Phone Book	
Name	Telephone#
Bob	416-212-1234
Sara	905-213-1111
Jack	416-433-1322

Table 3.7: A Row Table for checking entries in a phone book

Phone Book	
Name	Telephone#
Bob	416-212-1234
Sara	905-213-1111
Jack	416-433-1322

Table 3.8: Result of Table 3.7

The binding of the table headings, i.e., *"Name"* and *"Telephone#"*, to the associated agents are done in the make routine as before (lines 10–14).

```
 1 class
 2   TELEPHONE_ROW_FIXTURE
 3 inherit
 4   ES_ROW_FIXTURE [TELEPHONE_ENTRY]
 5
 6 create
 7   make
 8
 9 feature
10   make is -- Binding is done in here
11     do
12       bind("Name", agent get_name)
13       bind("Telephone#", agent get_number)
14     end
15
16   get_name(an_element: TELEPHONE_ENTRY): STRING is
17     do Result := an_element.name end
18
19   get_number(an_element: TELEPHONE_ENTRY): STRING is
20     do Result := an_element.telephone_number end
21
22   query (list: STRING): LINKED_LIST[TELEPHONE_ENTRY] is
23     local
24       elem1, elem2, elem3: TELEPHONE_ENTRY
25     do
26       create elem1.make ("Bob", "416-212-1234")
27       create elem2.make ("Sara", "905-213-1111")
28       create elem3.make ("Jack", "416-433-1322")
29       create Result.make
30       Result.extend (elem1)
31       Result.extend (elem2)
32       Result.extend (elem3)
33     end
34 end
```

Figure 3.6: The Fixture code associated with the Action Table 3.7

The query at line 22 is a deferred feature inherited from the ES_ROW_FIXTURE which must be effected in the subclass. For this simple example, the business logic resides in the Fixture code, e.g., the database of telephone entries is created and populated in the Fixture code (lines 26–32); however, in complex systems, the job of the `query` feature is to retrieve the database from the system under test and convert such database to a `LINKED_LIST[G]` where `G` is the generic type of the objects defined in the Row Fixture (`TELEPHONE_ENTRY` in our example).

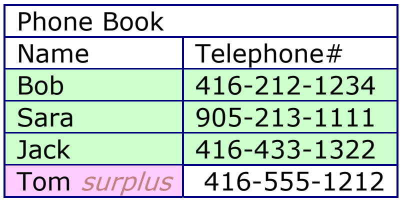
The result of running Table 3.7 is shown in Table 3.8. A green row indicates that the element described by the row matches an element in the database (e.g., in the system under test, there exists a telephone entry object whose name is *"Bob"* and with telephone number *"416-212-1234"*). If there are some objects in the database of the system under test which are not expected (i.e., there are no corresponding rows in the table), then they will be reported as *Surplus*. For example, if we extend the database with a new telephone entry item (with name *"Tom"* and telephone number *"416-555-1212"*), then the resulting table will look like Table 3.9. On the other hand, if some elements are expected in the table (in the form of rows) but the database does not contain objects corresponding to those rows, they will be reported as *Missing*. For example, if we remove line 31 in Fig. 3.6, the database will miss the telephone entry associated with *"Sara"*; therefore, the resulting table after execution of ES-Fit will look like Table 3.10. The code of class `TELEPHONE_ENTRY` (part of the business logic) is shown in Fig. 3.7.

```
class
    TELEPHONE_ENTRY
create
    make

feature -- Creation

    make (a_name: STRING; a_number: STRING) is
        require
            name_non_void: a_name /= void
            number_non_void: a_number /= void
        do
            name := a_name
            telephone_number := a_number
        ensure
            name_set: name.is_equal (a_name)
            number_set: telephone_number.is_equal (a_number)
        end

feature -- Implementation

    name: STRING

    telephone_number: STRING
end
```

Figure 3.7: A `TELEPHONE_ENTRY` object with `name` and a `telephone_number`

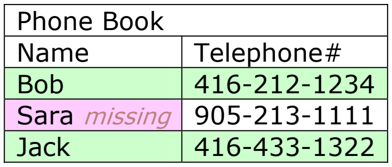
3.3 Errors in Fit Tables

In Chapter 2, Scenario Tests and ML-Contracts helped us to specify aspects of the design in an automatically testable format. When these tests run successfully, we obtain a certain amount of confidence that the implementation satisfies the specification. However, there is yet no guarantee that the specified design satisfies the requirements as described in the Fit tables. We may be designing the product right—yet, we still do not know if we have the right product! There



Phone Book	
Name	Telephone#
Bob	416-212-1234
Sara	905-213-1111
Jack	416-433-1322
Tom *surplus*	416-555-1212

Table 3.9: Database contains an unexpected element



Phone Book	
Name	Telephone#
Bob	416-212-1234
Sara *missing*	905-213-1111
Jack	416-433-1322

Table 3.10: Database misses an expected element

are two types of errors that may be reported by ES-Fit:

- A *category 1* error is one in which the expected behaviour declared in the table disagrees with the actual behaviour of the system (without generating any contractual violations). This type of error indicates that the implementation is faulty and (in addition) that there is either a wrong or incomplete specification (that was not flagged by an appropriate contract violation).

- Any contract violation is a *category 2* error and such violations are reported in the Fit tables. This type of error indicates that implementation does not satisfy its specification (contract).

Consider a customer who *requires* storage space on their system of 180 Gigabytes as shown in Table 3.11. This requirement may be written as $storage \geq 180GB$. A variety of specifications are shown together with an incorrect implementation ($make := Seagate$; $storage := 160GB$).

The first specification ($make \in \{Seagate, Hitachi\} \wedge storage = 160GB$) is wrong because it specifies too little storage to satisfy the requirement. The second specification ($make \in \{Seagate, Hitachi\}$) is incomplete because while specifying the make of the hard drive, it omits to specify the size (thus, allowing an implementor to provide too little storage). The third specification is both correct and complete (satisfies the requirements) and thus catches the incorrect implementation.

- Requirement: $storage \geq 180GB$
- **Specification:** see the table below
- Implementation: $make := Seagate\ ;\ storage := 160GB$

SPECIFICATION	**CATEGORY 1** (EXPECTED VS. ACTUAL)	**CATEGORY 2** (CONTRACT VIOLATION)
make $\in$ *{ Seagate, Hitachi}* $\wedge$ *Storage = 160GB*	✗ *Specification Error*	
make $\in$ *{ Seagate, Hitachi}*	✗ *Incomplete Specification*	
make $\in$ *{ Seagate, Hitachi}* $\wedge$ *Storage = 200GB*		✗ *Implementation Error*

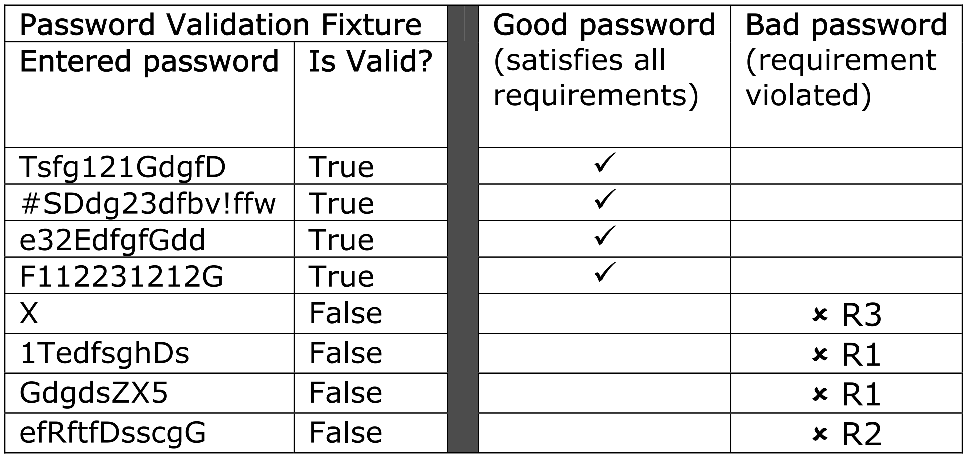
Table 3.11: Fit table errors

3.3.1 Password Example

Let's suppose that we are implementing a password validation module for an online banking service. The password rules are described as follows:

- **[R1]** A password must contain a non-numeric character in the first and last position
- **[R2]** A password must contain at least one capital alphabetic and one non-alphabetic character

- **[R3]** A password must be at least six characters in length



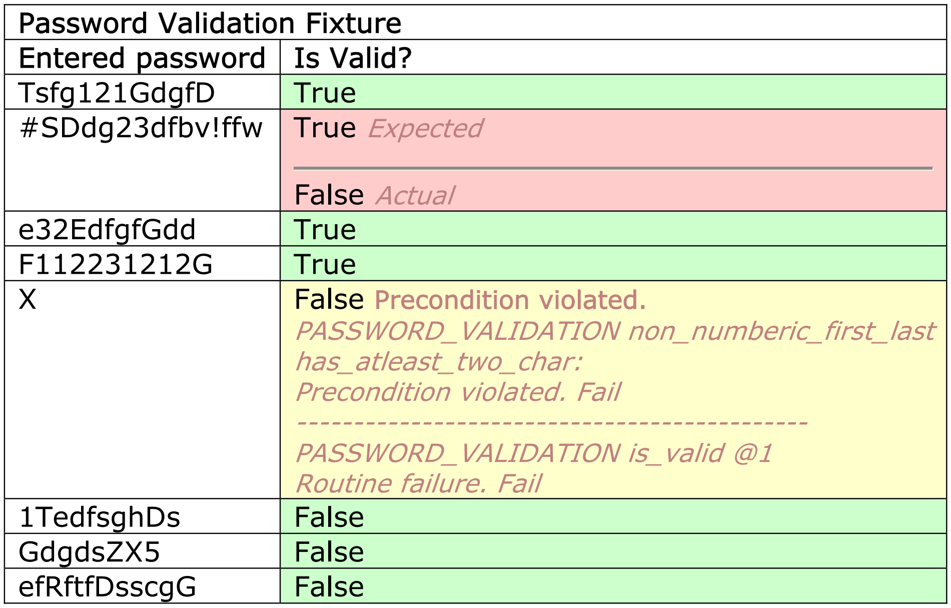
Password Validation Fixture			Good password (satisfies all requirements)	Bad password (requirement violated)
Entered password	Is Valid?			
Tsfg121GdgfD	True		✓	
#SDdg23dfbv!ffw	True		✓	
e32EdfgfGdd	True		✓	
F112231212G	True		✓	
X	False			× R3
1TedfsghDs	False			× R1
GdgdsZX5	False			× R1
efRftfDsscgG	False			× R2

Table 3.12: Traceability of the informal requirements to Fit requirements

Consider the Fit Table 3.12 providing checks of requirements R1 to R3 in which our customer has provided us with a series of concrete examples of valid and invalid passwords. We effectively have two "databases" of the functional requirements R1 to R3: (a) the informal requirements and (b) the Fit tests. It is important to relate the two so that we can check that all the informal requirements are formally checked by Fit.

Tracing the informal requirements to the Fit tests is complicated by the fact that there is a many-to-many relationship between the informal requirements and the Fit tests (both ways). In developing Fit tests it is important to show that all the requirements are covered and that there is consistency between the two databases.

The same kind of consistency check can be made using the UML's use case notation. However, the advantage of Fit tests is that we will actually be able to check that these tests pass.

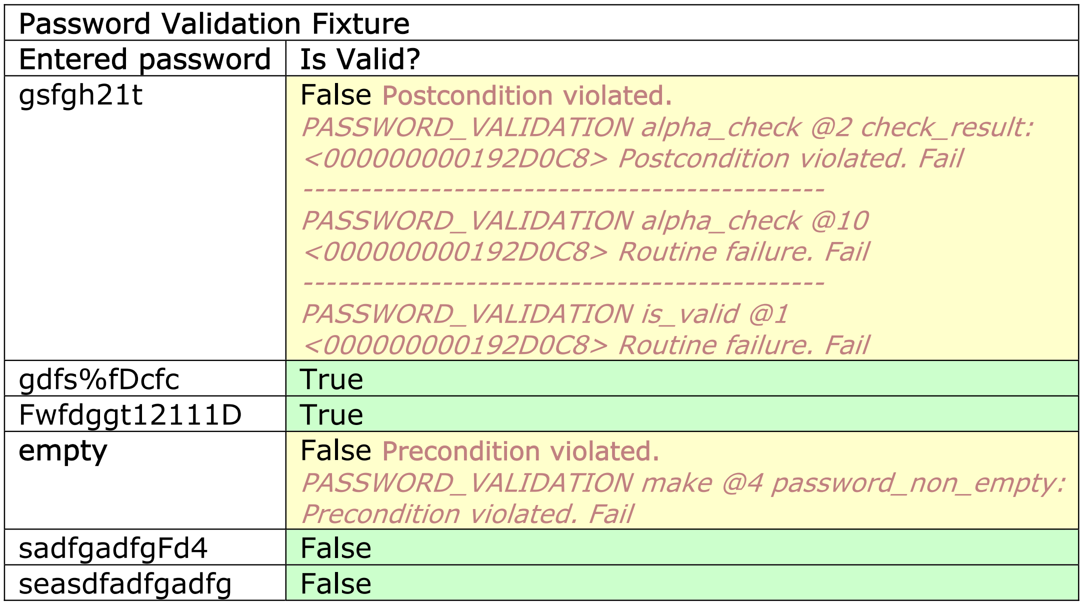


Password Validation Fixture	
Entered password	Is Valid?
Tsfg121GdgfD	True
#SDdg23dfbv!ffw	True *Expected* False *Actual*
e32EdfgfGdd	True
F112231212G	True
X	False Precondition violated. *PASSWORD_VALIDATION non_numberic_first_last* *has_atleast_two_char:* *Precondition violated. Fail* *---* *PASSWORD_VALIDATION is_valid @1* *Routine failure. Fail*
1TedfsghDs	False
GdgdsZX5	False
efRftfDsscgG	False

Table 3.13: Column table for password validation module

In order to check the requirements, the tables are connected to the password validation module `PASSWORD_VALIDATION` (shown in Fig. 3.13 and 3.14) through a Column Fixture. The Fixture initializes a password validation object and calls the `is_valid` feature on that object substituting the password provided in the table.

After running the tables against the implementation, the following tables are generated by ES-Fit (Tables 3.13 and 3.14). The forth row of Table 3.13 shows



Password Validation Fixture	
Entered password	Is Valid?
gsfgh21t	False Postcondition violated. *PASSWORD_VALIDATION alpha_check @2 check_result: <00000000192D0C8> Postcondition violated. Fail* *---* *PASSWORD_VALIDATION alpha_check @10 <00000000192D0C8> Routine failure. Fail* *---* *PASSWORD_VALIDATION is_valid @1 <00000000192D0C8> Routine failure. Fail*
gdfs%fDcfc	True
Fwfdggt12111D	True
empty	False Precondition violated. *PASSWORD_VALIDATION make @4 password_non_empty: Precondition violated. Fail*
sadfgadfgFd4	False
seasdfadfgadfg	False

Table 3.14: Continuation of table 3.13

an example of an expected vs. actual error (category 1) which does not involve a contract violation. As we mentioned before, this type of failure indicates that there is something wrong in the specification. In our case, the postcondition of the routine non_numberic_first_last (Fig. 3.8) has a problem.

```
non_numberic_first_last: BOOLEAN is
  require
    password.count >= 2
  ensure
    Result = password[1].is_alpha and password[count].is_alpha
  end
```

Figure 3.8: Contracts of non_numberic_first_last

The postcondition asserts that the routine returns true iff both the first and the last characters of the password are alphabetic (`is_alpha` is a feature of `CHARACTER` class which returns true if the character is alphabetic). The informal requirements asserted that a password must have a "non-numeric character in the first and last position". Since a non-numeric character is not necessarily an alphabetic one (e.g., '#') we can conclude that the postcondition was wrong. The implementation also needs to be revised to satisfy the new postcondition.

If we make the adjustments, the Fit table will indeed show green for that case. The revised version of the routine is shown in Fig. 3.9.

```
non_numberic_first_last: BOOLEAN is
  require
    has_atleast_two_char: password.count >= 2
  do
    Result := not (password.item (1).is_digit or
    password.item (password.count).is_digit)
  ensure
    check_result: Result = not (password[1].is_digit or password[count].
        is_digit)
  end
```

Figure 3.9: Fixed version of routine `non_numberic_first_last`

Now we can focus on the next error in Table 3.13 (where a short password *"X"* is provided). A *precondition violation* is reported with an error message pointing us to the location of the violation (feature `non_numberic_first_last`). This type of violation is an indication of an implementation problem in the body of the routine that is the client of `non_numberic_first_last`. The error trace points us to is_valid as the caller. Closer look at the implementation (lines 17–21 in Fig. 3.13) reveals that is_valid makes a call to non_numberic_first_last without satisfying its precondition password.count >= 2. A string with size less than 2 (e.g. *"X"*) caused this precondition violation. The revised version of is_valid is shown in Fig. 3.10. This implementation implicitly checks for the precondition of the non_numberic_first_last (through lazy evaluation). As a result of this code change, all the tests in table 3.13 will pass.

```
is_valid: BOOLEAN is
  do
    Result := (password.count >= Valid_length and then
    non_numberic_first_last and alpha_check)
  end
```

Figure 3.10: Fixed version of is_valid query

We can now focus on Table 3.14. Both of the failures reported in this table involve a contract violation indicating a category 2 error. The first one (3rd row) involves a *postcondition violation*. A postcondition violation tells us that there is an implementation problem somewhere in the code, i.e., the implementation does not satisfy its specification. In our case, the reported error message in the table indicates that the postcondition of routine alpha_check is violated at the second line of the postcondition (line 42 in Fig. 3.13).

According to this postcondition, the routine returns true iff the password contains at least one capital alphabetic and one non-alphabetic character. Since the postcondition was violated on input password (*"gsfgh21t"*), we conclude that the routine is not properly implemented to do the check for **capital** alphabetic characters (i.e., line 32 in Fig. 3.13 should also check if the character is uppercase). The fixed version of the routine is shown in Fig. 3.11.

```
alpha_check: BOOLEAN is
  require
    non_empty: not password.is_empty
  local
    i: INTEGER; r1, r2: BOOLEAN
  do
    from i := 1
    until Result or i > password.count
    loop
      if (password.item (i).is_alpha and
        password.item (i).is_upper) then
        r1 := true
      elseif not password.item (i).is_alpha then
        r2 := true
      end
      i := i + 1
      Result := r1 and r2
    end
  ensure
    check_result: Result =
    (to_list.there_exists (agent is_alpha_caps(?))
    and to_list.there_exists (agent is_non_alpha(?)))
  end
```

Figure 3.11: Fixed version of routine `alpha_check`

This brings us to the last violation in table 3.14 which involves a *precondition violation*. The precondition of routine `make` is violated (line 10 in Fig. 3.13). This precondition specifies that the `make` routine expects a non-empty string as the input argument. The customer expects that the system deals with empty passwords and rejects them[6].

This type of error in the Fit table is a special case indicating a *specification error* made by the programmers (i.e., the front modules to the outside world must have been designed in a defensive way to be able to deal with unexpected input values from the customers). Thus, the password validation module must be able to deal with empty passwords as well as non-empty ones. The solution is to remove the precondition as shown in Fig. 3.12. After this fix, both tables will indeed pass.

```
make (a_password: STRING) is
     require
         password_non_void: a_password /= Void
     do
         password := a_password
     ensure
         password_set: password.is_equal (a_password)
     end
```

Figure 3.12: Fixed version of routine `make`

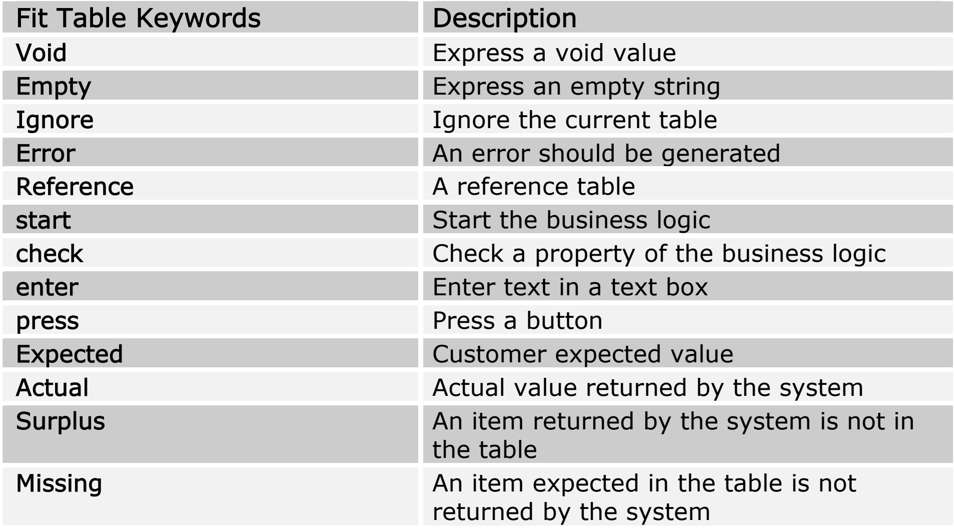
3.3.2 Error keyword

Fit also allows the developers to describe a case in which the code should fail with the use of **error** keyword in the table. This is very similar to a violation test case (Chapter 2).

[6] This is asserted by the use of “empty” keyword in the Fit table.

As an example, let's assume that the precondition error in Table 3.14 was indeed expected from the point of view of the developer, i.e., the developer expects that the code should fail on empty passwords.

In order to describe such a case, we use the **error** keyword. By declaring the expected value of a cell to be "error", we assert that an exception should be generated. If such an exception happens, the Fit row will pass. Table 3.15 shows a summary of other keywords that are available in Fit tables.



Fit Table Keywords	Description
Void	Express a void value
Empty	Express an empty string
Ignore	Ignore the current table
Error	An error should be generated
Reference	A reference table
start	Start the business logic
check	Check a property of the business logic
enter	Enter text in a text box
press	Press a button
Expected	Customer expected value
Actual	Actual value returned by the system
Surplus	An item returned by the system is not in the table
Missing	An item expected in the table is not returned by the system

Table 3.15: List of keywords allowed in the Fit tables

3.4 Reference Tables

ES-Fit extends the original Fit framework by adding the notion of "Reference" Tables. Section 5.1.11 explains the details regarding the Reference Tables.

3.5 Summary

In this chapter, we showed how to capture customer requirements through the use of Fit tables. We introduced three types of Fit tables and their associated Fixture code and discussed various contract violations that could be reported to these tables. In the next chapter, we use an example to show how our method can be used to detect bugs early in the development process.

```
class PASSWORD_VALIDATION create
  make
feature
  password: STRING
  Valid_length: INTEGER is 6

  make (a_password: STRING) is
    require
      password_non_void: a_password /= Void
      password_non_empty: not a_password.is_empty
    do
      password := a_password
    ensure
      password_set: password.is_equal (a_password)
    end

  is_valid: BOOLEAN is
    do
      Result := (non_numberic_first_last and
      alpha_check and password.count >= Valid_length)
    end

  alpha_check: BOOLEAN is
    require
      non_empty: not password.is_empty
    local
      i: INTEGER; r1, r2: BOOLEAN
    do
      from i := 1
      until Result or i > password.count
      loop
        if password.item (i).is_alpha then
        r1 := true
        elseif not password.item (i).is_alpha then
        r2 := true
      end
      i := i + 1
      Result := r1 and r2
    ensure
      check_result: Result =
      (to_list.there_exists (agent is_alpha_caps(?)) and
      to_list.there_exists (agent is_non_alpha(?)))
    end
```

Figure 3.13: Password Validation Module

```
  non_numberic_first_last: BOOLEAN is
    require
      has_atleast_two_char: password.count >= 2
    do
      Result := (password.item (1)).is_alpha and
      password.item (password.count).is_alpha
    ensure
      check_result: Result = (to_list.first.is_alpha and to_list.last.is_alpha)
    end

feature -- Support for contracts
  to_list: LINKED_LIST [CHARACTER] is
      -- converts 'password' to list for contracts
    local
      i: INTEGER
    do
      from create Result.make; i := 1
      until i > password.count
      loop
        Result.force (password.item (i))
        i := i + 1
      end
    end

  is_alpha_caps (c: CHARACTER): BOOLEAN is
    do Result := c.is_alpha and c.is_upper end

  is_non_alpha (c: CHARACTER): BOOLEAN is
    do Result := not c.is_alpha end

invariant
  password_non_void: password /= Void
end -- End of class PASSWORD_VALIDATION
```

Figure 3.14: Password Validation Module, Cont.

4 A Case Study

In this chapter, we illustrate our method of early testable requirements and specifications with a chat room example. The point of the example is to show that our approach and tool is wide-spectrum (i.e., deals both with customer requirements and design specifications) in an integrated fashion. We provide a brief overview before delving into the details.

4.1 Overview

We start with a description of the problem domain. The phenomena of the problem domain includes entities such as chat rooms, users, messages, the chat room administrator and the interactions between these entities. The informal requirements refer to the phenomena of the problem domain and are normally stated as English text, e.g., "allow chat users to connect or disconnect from the chat server" or "the chat server may move a user from one room to another".

We would like to convert these informal requirements into testable requirements. To do this, customers and developers translate as much of the English text as possible into Fit tables which captures the nature and interaction of the entities in the chat application in a testable format. For example, our first Fit requirement for the chat application is provided in Table 4.1.

At this point, there is no actual code that implements the application—thus, all the Fit tests fail. The job of the developer is to describe a design that satisfies the requirements. The design is expressed in terms of the phenomena of the solution space (i.e., the phenomena of the machine in Jackson's terminology, see Chapter 1). So, for example, the design may be expressed in terms of object-oriented classes such as CHAT_SERVER, CHAT_ROOM, and CHAT_USER. Features such as enter_room will be needed in class CHAT_SERVER to move a user from one chat room to another. An initial design is shown in the BON diagram of Fig. 4.3.

A design is one thing. Knowing that the design satisfies the requirements is something else. We could check an implementation of the design directly against the Fit tables. A Fit table failure would then indicate a flaw in the implementation of the design.

However, design implementations are usually complex. Feature enter_room (in class CHAT_SERVER) which moves a user from one chat room to another may need a search algorithm to find the appropriate user to be moved. Thus, we may need to write a search routine that is not immediately checked by the Fit tables. How will we check that this routine works correctly before integrating it into feature `enter_room`? Even after it is integrated, it may work correctly in the cases described by the Fit tables, yet have flaws in cases not checked by the tables. Also, the feature `enter_room` may not correctly interact when used in conjunction with other features. There are usually an enormous number of interactions between components of the design that may go wrong and that need to be checked. This is why developers test implemented components either as they are developed or *post facto*, irrespective of any acceptance tests that will be conducted by their customers.

But, against what must the implementation be tested? The answer is that the developer has an intuitive idea how the design must behave—this is the design *specification*. Specifications are not the same thing as the customer requirements (that describe phenomena in the problem domain). Specifications must describe the phenomena in the solution space such as class `CHAT_SERVER`, its feature `enter_room` and the search routine that it depends upon.

A design specification must also manifest good design principles. Some important ideas in this context include the notions of information hiding and separation of concerns via well-defined module interfaces. A large program should be divided into separate chunks that can be developed and tested independently of each other:

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. ... Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching ([78], quoting Gauthier and Pont "Designing Systems Programs").
>
> Parnas's information-hiding definition of modules is the first published step in [a] crucially important research program, and it is an intellectual ancestor of object-oriented programming. He defined a module as a software entity with its own data model and its own set of operations. Its data can only be accessed via one of its proper operations. The second step was a contribution of several thinkers: the upgrading of the Parnas module into an abstract data type, from which many objects could be derived. The abstract data type provides a uniform way of thinking about and specifying module interfaces, and an access discipline that is easy to enforce ([39], Chapter 19).

In fact (and in short), object-orientation = abstract data types + inheritance [84]. What form should the design specification take, given that we want the specification to be testable and a specification of the underlying abstract data types? We believe that ML-Contracts and Scenario Tests (see Chapter 2) are good candidates for such testable specifications.

Contracts document the complete behaviour of classes using class invariants and pre and post conditions for each feature in the class. As described in earlier chapters, the base Eiffel libraries of mutable classes are not adequate for complete specifications. This is where the (immutable) mathematical libraries (ML) described in Chapter 2 and Appendix C are needed. For example, using the ML-

Contracts we may model the chat application with: (a) a *location model* which is a function `ML_MAP[CHAT_USER,CHAT_ROOM]` (i.e., given a user, the map returns the current room of the user); (b) an *ownership model* which is a map from chat rooms to chat users. With these mathematical models we can provide complete contracts for classes such as `CHAT_SERVER` and features such as `enter_room` (e.g., see Fig. 4.17). This will be explained in detail in the sequel.

Once designs are specified with contracts, we can check that the implementation of the chat room design satisfies the contracts (i.e., the specification). In Chapter 1, we called this implementation correctness. The check of implementation correctness can be done statically via theorem provers or dynamically via runtime assertion checking.

Implementation correctness checks that the implementation satisfies the design specification. However, we do not yet know that we have designed the right product, one that satisfies the goals of the customer. What we must do is check specification correctness (as described in Chapter 1), i.e., the specification must satisfy the customer requirements (Fit tables). Our software assurance tool ESpec does this by reflecting contract violations in the Fit tables (for an example, see Table 4.2 in the sequel). Such violations may lead us to detect mismatches between implementation, specifications, and the requirements.

As shown in Fig. 4.1, we are not proposing a software development method-

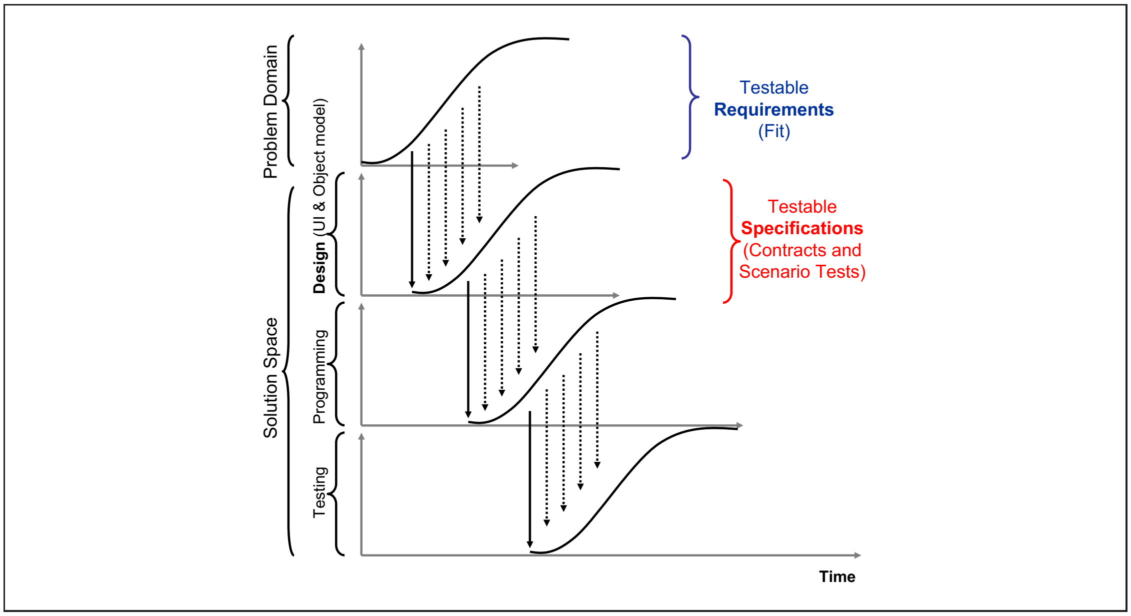


Figure 4.1: Concurrent Design

ology (also see Chapter 1). The figure indicates that requirements elicitation, design, programming and testing may be done concurrently. What we are proposing is a method that allows us to test requirements and specifications no matter where we are in the design process. Thus, we could start with Fit tables, ML-Contracts, and then implementation and testing. Or we could start with implementations and then write the specifications and requirements.

4.2 Informal Requirement Document

Our customer needs a *chat application* that allows company employees to communicate with each other. The chat application should allow members to post messages to one or more of the authorized list of people who are currently online.

The entities in the problem domain include many *chat users* and a *server administrator* of the *chat server*. Users can join permitted rooms and send messages to the other users of those rooms. A user who first creates a room is the *owner* of the room and has the power to decide who accesses that room. A user who is forbidden from a room, if currently in that room, must be moved to the "Lobby". In summary, the application has the following requirements:

R1: A chat server has an "administrator" and a room called the "Lobby". After starting the server, the administrator should be in the Lobby. Initially, the administrator is the only user and the Lobby the only room

R2: A user may connect to the chat server thus landing in the Lobby or may disconnect from the server

R3: A user may add or remove public or private rooms thus becoming the owner of the room

R4: An owner may permit or reject other users from accessing rooms

R5: A user may enter or exit rooms as allowed by the owner of the room

R6: After entering an allowed room, a user may read and post messages in the room

Chat rooms need to be listed by their unique names so that the users can find sessions relevant to their interests. Users should not be able to see the listings of private rooms unless they are permitted to do so. Public rooms should be visible to everyone.

4.3 A Testable Requirement Document (Fit table)

We (the customer and/or developer) are ready to write an HTML requirements document with appropriate Fit table formalizations. We start with the first requirement:

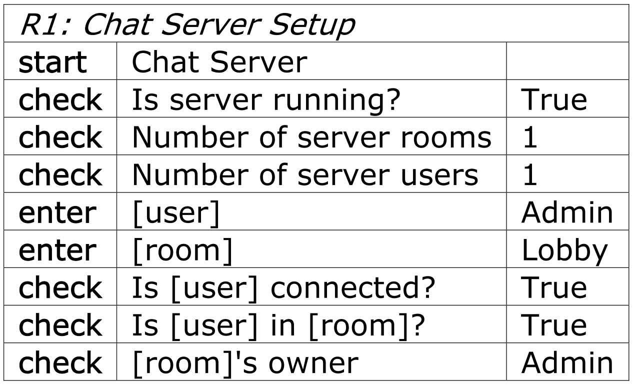
4.3.1 The Customer's First Fit Table

> **R1:** A chat server has an administrator and a room called the Lobby. After starting the server, the administrator is in the Lobby. Initially, the administrator is the only user and the Lobby the only room.

The first Fit table, shown in Table 4.1, is an *Action Table*. This table has a title "R1: Chat Server Setup" and starts a chat server after which various properties relating to R1 are checked.

An Action Fixture is a developer written class (e.g., `CHAT_ACTION`) that will be used to glue the implementation (the business logic) to the customer written Fit table (Table 4.1). The title in the Fit table ("R1: Chat Server Setup") will be used to bind the customer Fit table to the execution of developer written Fixture class

- Start the chat server.
- Check that the chat server is up and running.
- Check that there is one room (the Lobby).
- Check that there is one user (the Administrator).
- Set [user] to “Admin” and [room] to “Lobby”.
- Check that [user] “Admin” is connected and in [room] “Lobby”.
- Check that the owner of the “Lobby” is “Admin”.



R1: Chat Server Setup		
start	Chat Server	
check	Is server running?	True
check	Number of server rooms	1
check	Number of server users	1
enter	[user]	Admin
enter	[room]	Lobby
check	Is [user] connected?	True
check	Is [user] in [room]?	True
check	[room]'s owner	Admin

Table 4.1: Chat Action Table for requirement R1

CHAT_ACTION. The title is selected by the customer. We first describe the Action Table; later in the sequel, we will return to the details of the Action Fixture glue code.

As described in Chapter 3, there are four keywords that may be used by a customer in an Action Table: **start**, **check**, **enter** and **press**. From the Eiffel Fixture code point of view, we may think of “**start** x” as creating an object attached to entity x. We may think of “**check** q v” as checking that query q has value v.

Finally, the **enter**/**press** combination is used to call a routine "$r(v_1, v_2)$", where v_1 and v_2 are the set by **enter** and **press** is used to invoke r.

Consider the check for the property "Is [user] in [room]?" in row 9 of Action Table 4.1. We could have used the descriptive string "Is Admin in Lobby?" for the property. However, that limits this description to the specific property involving the specific individuals Admin and Lobby. We would prefer to check for the more generic property that some arbitrary user is in a given room. We use the keyword **enter** to associate a value with a parameter of the property (like an argument of a query). Thus, at row 6, the customer associates the value "Admin" with the parameter "[user]". The customer could have chosen "some user" rather than "[user]" in the second column or some other descriptive string. We use the convention of surrounding the parameter with square brackets so that it stands out as a parameter of the property, e.g., in the property "Is [user] in [room]?" at line 9 the parameters are "[user]" and "[room]" entered at lines 6 and 7 respectively.

The keyword **press** is not used in Table 4.1 but it will be used in the sequel. This keyword denotes an action (like pressing a button) that effects some change in the business logic. The keyword **press** may be used together with **enter** to denote a parameterized action, e.g., we may use **press** together with the parameterized action "[user] adds [room]" as in Action Table in Fig. 4.11. This means that user “Bob” adds the room “Technical Support” to the chat application, and “Bob” is now the owner of the room.

Fit tables make the requirements testable. However, at this point, if we run the Fit table in the requirements document it will fail. For example, the checks associated with the value cells in Table 4.1 will display as red indicating that the requirement is not yet satisfied. As yet there is no implementation code and so we expect failure. Our goal is now to specify a *design* that will satisfy the requirements (i.e., cause each test row in the table to pass).

4.4 Test Driven Design or Design by Contract?

How do we add the new functionality required by Fit tables such as the one provided in the previous section? We could follow Test Driven Design Development by writing unit tests for the new functionality, or we could follow a Design by Contract approach by writing contracts for the features of the chat server. Both approaches are good ways to specify the Design.

In the next section, we illustrate the use of design specifications following a Design by Contract approach. In the rest of this section we briefly outline the Test Driven approach.

The Scenario Test in Fig. 4.2 specifies part of the design needed to satisfy the Fit table provided in the previous section. The Scenario Test describes a col-

```
scenario_test: BOOLEAN is
     local
       server: CHAT_SERVER
       mike, anna: CHAT_USER
       mike_room: CHAT_ROOM
       users: LIST[CHAT_USER]
       rooms: LIST[CHAT_ROOM]
    do
       -- create the chat server and check it
       create server.make
       users := server.users
       rooms := server.rooms
       check server.user_count = 1 end
       check server.room_count = 1 end

       -- create 2 users Mike and Anna and connect them to the server
       create mike.make ("Mike")
       create anna.make ("Anna")
       server.connect (mike)
       server.connect (anna)
       check server.user_count = 3 and server.room_count = 1 end
       check mike.room = server.lobby and anna.room = server.lobby end
       check users.has(mike) and users.has(anna) end

       -- Mike creates and adds a room ‘‘Technical Support"
       mike_room := mike.create_room ("Technical Support")
       mike.add_room (mike_room)
       check server.room_count = 2 end
       check not mike_room.is_private end
       check rooms.has(mike_room) end

       -- Mike changes the status of his room to private
       mike.set_private ("Technical Support")
       check mike_room.is_private end
       check not server.is_allowed (anna, "Technical Support") end

       -- Mike allows Anna to join the Technical Support room
       mike.allow_user ("Anna", "Technical Support")
       check server.is_allowed (anna, "Technical Support") end
       Result := True
    end
```

Figure 4.2: Scenario Test for a chat server

laboration between three classes CHAT_SERVER, CHAT_USER, and CHAT_ROOM. It thus specifies the design shown in the BON diagram in Fig. 4.3.

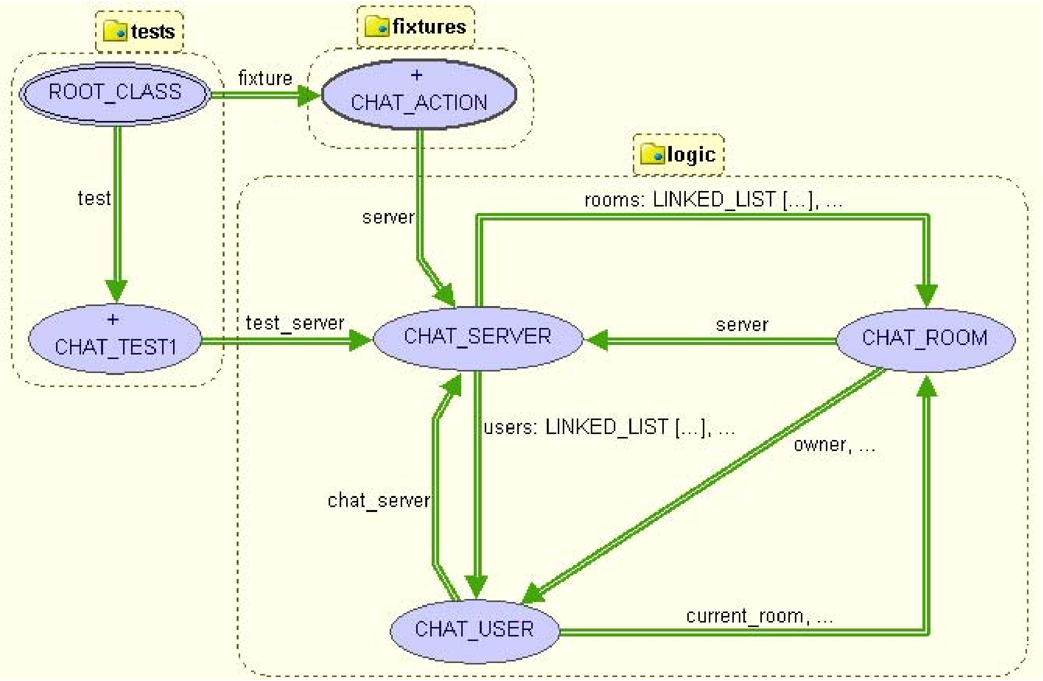


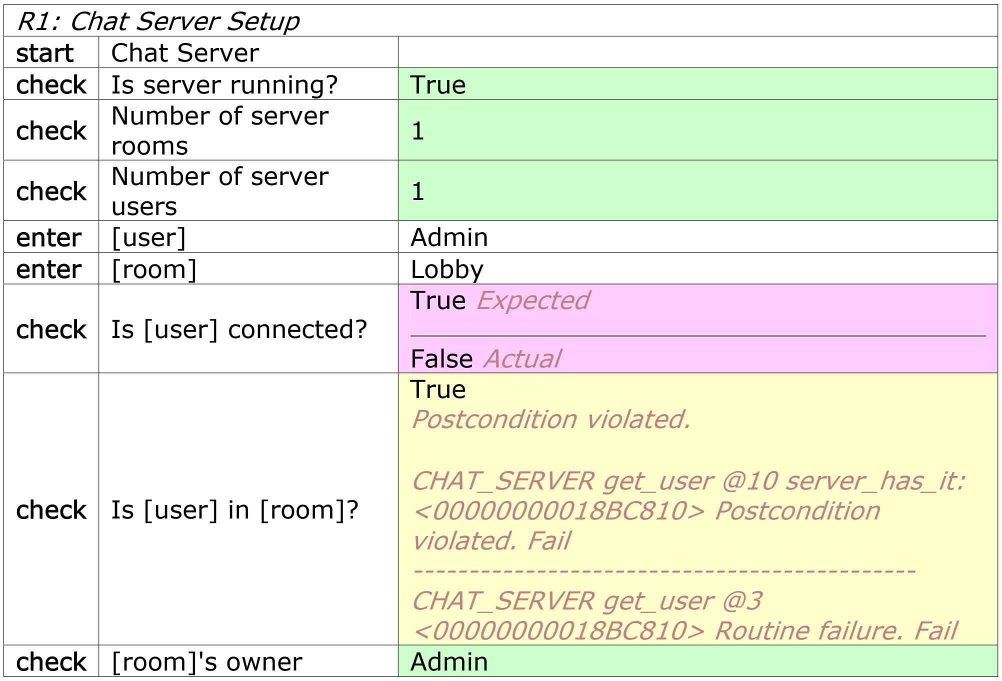
Figure 4.3: Design of the chat application as a BON class diagram

The test specifies specific features in CHAT_SERVER such as:

- users: LIST[CHAT_USER]
- rooms: LIST[CHAT_ROOM]
- add_room (a_user: CHAT_USER)

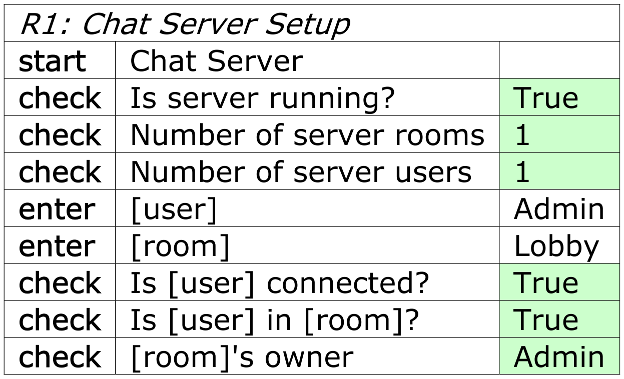
If all the classes and features in the Scenario Test are added, the project will compile and the design illustrated in the BON class diagram in Fig. 4.3 is generated automatically. The class diagram presents the design so far (classes and feature signatures, but not yet code in the bodies of the features).

Getting this Scenario Tests to compile (thus producing the design in Fig. 4.3) and pass (by adding implementation code) is an important step towards getting the Fit table to succeed (see Section B.3 for more tests).



R1: Chat Server Setup		
start	Chat Server	
check	Is server running?	True
check	Number of server rooms	1
check	Number of server users	1
enter	[user]	Admin
enter	[room]	Lobby
check	Is [user] connected?	True *Expected* False *Actual*
check	Is [user] in [room]?	True *Postcondition violated.* *CHAT_SERVER get_user @10 server_has_it: <00000000018BC810> Postcondition violated. Fail* *--* *CHAT_SERVER get_user @3 <00000000018BC810> Routine failure. Fail*
check	[room]'s owner	Admin

Table 4.2: Result of running Table 4.1 shows contract errors.



R1: Chat Server Setup		
start	Chat Server	
check	Is server running?	True
check	Number of server rooms	1
check	Number of server users	1
enter	[user]	Admin
enter	[room]	Lobby
check	Is [user] connected?	True
check	Is [user] in [room]?	True
check	[room]'s owner	Admin

Table 4.3: Result of running Table 4.1 after fixing the business logic.

4.4.1 Specification Correctness

Scenario Tests helped us to specify aspects of the design in an automatically testable format. When these tests run successfully, we obtain a certain amount of confidence that the implementation satisfies the specification. However, there is yet no guarantee that the specified design satisfies the requirements as described in the Fit tables. We may be designing the product right—yet, we still do not know if we have the right product!

As we illustrated previously, if we run Action Table 4.1 with an incorrect implementation or design, we get error results of the kind shown in Table 4.2.

Two types of errors are shown in Table 4.2. The first error (row 8 shown in pink) indicates that the routine to check if a user is connected is not doing what was expected (the value “True” was expected but the actual value returned by the business logic was “False”). As we described in Chapter 3, We call this type of error a category 1 error. Category 1 errors usually indicate that the design was not specified correctly. The implemented code was correct in a sense that it did not trigger any contract violations or Scenario Test errors yet it failed to satisfy the customer requirements. The design specification (via Scenario Tests and contracts) is either incomplete or even incorrect.

The second error (row 9 shown in yellow) indicates a postcondition failure in the business logic (in `CHAT_SERVER.get_user`). This is an example of a category 2 error. Category 2 errors usually indicate an implementation problem in the business logic (i.e., the implementation failed to satisfy its contracts).

In a Test Driven approach, additional code is added and further tests developed (see Sections B.3 and B.4). This results in an implementation that passes all the Scenario Tests thus guaranteeing that the Fit table succeeds as shown in Fig. 4.3.

4.5 Writing Complete ML-Contracts

In the previous section we used Scenario Tests to write testable specifications. In this section we explore the use of contracts for writing testable specifications.

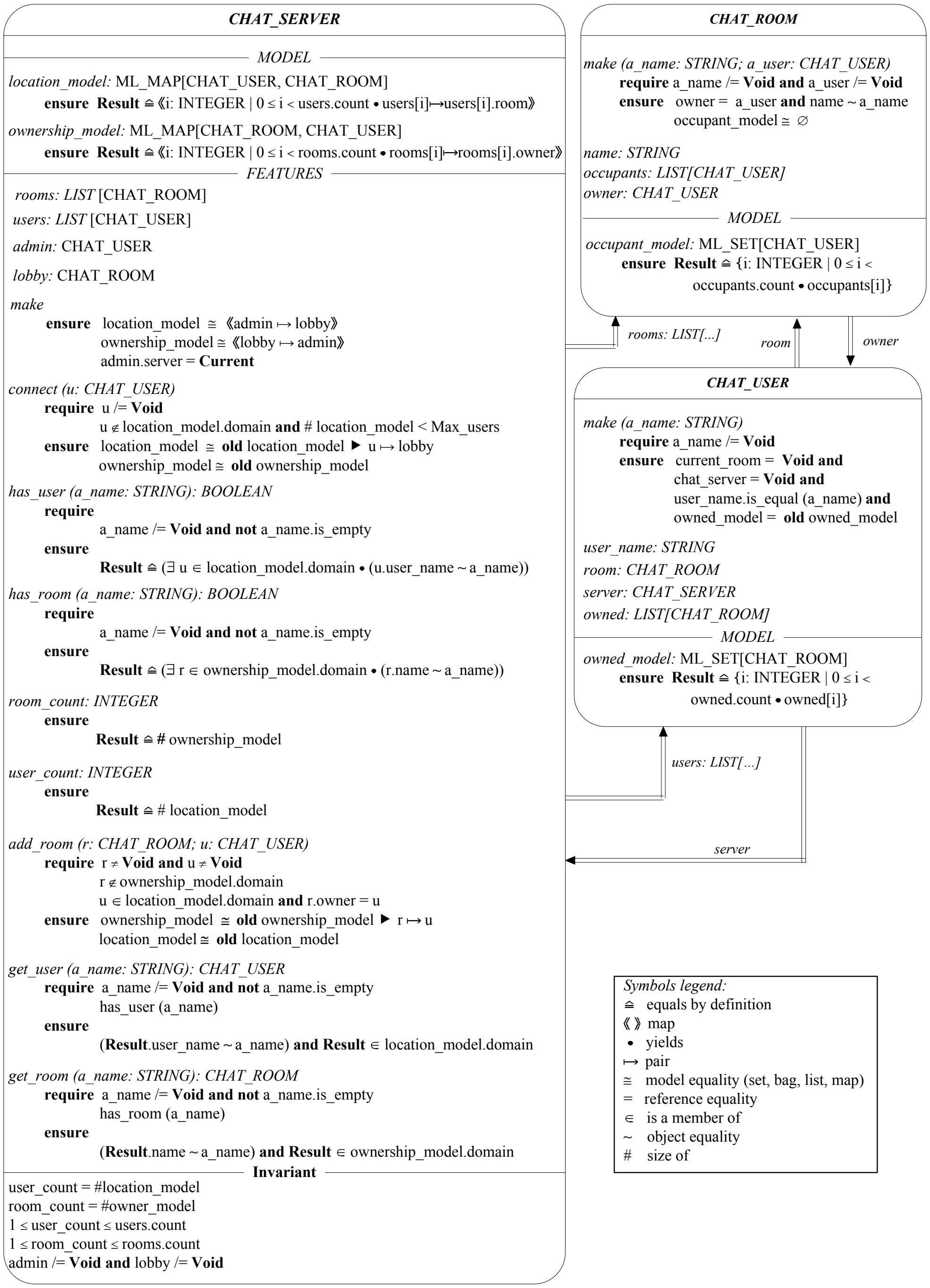
As we mentioned in Chapter 2, the basic contracting facilities of Eiffel language does not allow for complete contracts. We illustrate this lack and describe

```
class CHAT_SERVER ...

feature {NONE} -- private features

  rooms: LIST[CHAT_ROOM]
  users: LIST[CHAT_USER]

feature -- public features

  lobby: CHAT_ROOM
  admin: CHAT_USER

  connect (u: USER) is
    require
      a_user /= Void
      -- user u not already connected
    do
      ...
    ensure
      -- add u to the existing users in the lobby
    end
end
```

Figure 4.4: Incomplete contract for routine `connect`

the use of ML-Contracts.

Consider the contract for the routine `connect` in class `CHAT_SERVER` in Fig. 4.4. This feature allows a user `u` to connect to the server. A new user is not initially connected. In the precondition of routine `connect` we would like to specify that a new user `u` is not yet connected, i.e., is not yet in our list of users. In the postcondition, we would like to specify that the new user `u` is now added to the existing users of the Lobby.



```
CHAT_SERVER
—— MODEL ——
location_model: ML_MAP[CHAT_USER, CHAT_ROOM]
    ensure Result ≙《i: INTEGER | 0 ≤ i < users.count • users[i]⟼users[i].room》
ownership_model: ML_MAP[CHAT_ROOM, CHAT_USER]
    ensure Result ≙《i: INTEGER | 0 ≤ i < rooms.count • rooms[i]⟼rooms[i].owner》
—— FEATURES ——
 rooms: LIST [CHAT_ROOM]
 users: LIST [CHAT_USER]
 admin: CHAT_USER
 lobby: CHAT_ROOM
make
    ensure  location_model ≅ 《admin ↦ lobby》
            ownership_model ≅ 《lobby ↦ admin》
            admin.server = Current
connect (u: CHAT_USER)
    require u /= Void
            u ∉ location_model.domain and # location_model < Max_users
    ensure  location_model ≅ old location_model ▶ u ↦ lobby
            ownership_model ≅ old ownership_model
has_user (a_name: STRING): BOOLEAN
    require
            a_name /= Void and not a_name.is_empty
    ensure
            Result ≙ (∃ u ∈ location_model.domain • (u.user_name ~ a_name))
has_room (a_name: STRING): BOOLEAN
    require
            a_name /= Void and not a_name.is_empty
    ensure
            Result ≙ (∃ r ∈ ownership_model.domain • (r.name ~ a_name))
room_count: INTEGER
    ensure
            Result ≙ # ownership_model
user_count: INTEGER
    ensure
            Result ≙ # location_model
add_room (r: CHAT_ROOM; u: CHAT_USER)
    require r ≠ Void and u ≠ Void
            r ∉ ownership_model.domain
            u ∈ location_model.domain and r.owner = u
    ensure  ownership_model ≅ old ownership_model ▶ r ↦ u
            location_model ≅ old location_model
get_user (a_name: STRING): CHAT_USER
    require a_name /= Void and not a_name.is_empty
            has_user (a_name)
    ensure
            (Result.user_name ~ a_name) and Result ∈ location_model.domain
get_room (a_name: STRING): CHAT_ROOM
    require a_name /= Void and not a_name.is_empty
            has_room (a_name)
    ensure
            (Result.name ~ a_name) and Result ∈ ownership_model.domain
—— Invariant ——
user_count = #location_model
room_count = #owner_model
1 ≤ user_count ≤ users.count
1 ≤ room_count ≤ rooms.count
admin /= Void and lobby /= Void
```

```
CHAT_ROOM
make (a_name: STRING; a_user: CHAT_USER)
    require a_name /= Void and a_user /= Void
    ensure  owner = a_user and name ~ a_name
            occupant_model ≅ ∅
name: STRING
occupants: LIST[CHAT_USER]
owner: CHAT_USER
—— MODEL ——
occupant_model: ML_SET[CHAT_USER]
    ensure Result ≙ {i: INTEGER | 0 ≤ i <
            occupants.count • occupants[i]}
```

```
CHAT_USER
make (a_name: STRING)
    require a_name /= Void
    ensure  current_room = Void and
            chat_server = Void and
            user_name.is_equal (a_name) and
            owned_model = old owned_model
user_name: STRING
room: CHAT_ROOM
server: CHAT_SERVER
owned: LIST[CHAT_ROOM]
—— MODEL ——
owned_model: ML_SET[CHAT_ROOM]
    ensure Result ≙ {i: INTEGER | 0 ≤ i <
            owned.count • owned[i]}
```

Symbols legend:

≙	equals by definition
《 》	map
•	yields
↦	pair
≅	model equality (set, bag, list, map)
=	reference equality
∈	is a member of
~	object equality
#	size of

Figure 4.5: System Specifications in BON notation

How do we specify these contracts? One possibility is to use the private implementation data structures `users` and `rooms` which are linked lists of chat rooms and chat users (respectively) to write the contracts. This is not ideal because the implementation is low level and might change. We would like the specification of the feature to be independent of low level implementation details. In addition, not all classes are effective. Some classes are deferred (abstract) and thus there is no available implementation.

So the question is: how do we specify complete contracts without depending upon implementation detail?

4.5.1 The need for Mathematical Models

In order to fully specify the contracts of feature `connect` the chat application must remember:

1. All the users that are already connected (so that a check can be made that the same user does not connect twice).
2. All the users in the Lobby (so that the list of users of the Lobby can be updated when the new user is connected).

We may use a mathematical model to describe the above state of affairs. The *location model* is a function from `CHAT_USER` to `CHAT_ROOM` as shown in Fig. 4.6.

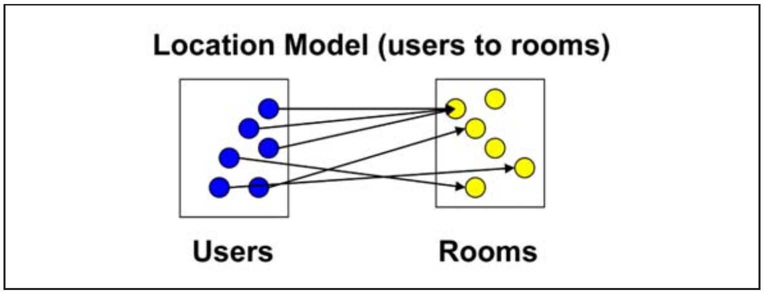


Figure 4.6: Location model—mapping from users to rooms

In the location model each user is associated with a room. The location model may be described using the mathematical map class ML_MAP as shown in Fig. 4.7.

In Fig. 4.7, the location model is specified as

```
location_model: ML_MAP[CHAT_USER,CHAT_ROOM]
```

The ML-precondition of routine `connect` is $u \notin location\_model.domain$ which asserts that the user u is not already connected (i.e., the user is not in the domain of the map). The ML-postcondition is

$$location\_model \quad \cong \quad (\textbf{old}\ location\_model) \blacktriangleright (u \mapsto lobby) \qquad (4.1)$$

which asserts that after execution of `connect`, the *location_model* is extended by (symbol ▶) the pair $u \mapsto lobby$, i.e., the location map in the poststate is the same as it was in the prestate but with the addition that user u is connected and in the Lobby. The symbol ≅ is the *model equality* symbol. Two maps are

(a) BON mathematical notation

```
class CHAT_SERVER feature

    location_model: ML_MAP[CHAT_USER,CHAT_ROOM]
    ownership_model: ML_MAP[CHAT_ROOM,CHAT_USER]

    lobby: CHAT_ROOM
    admin: CHAT_USER

    connect (u: USER) is
        require
            $u \neq Void \wedge u \notin location\_model.domain$
        ensure
            $location\_model \cong \texttt{old}\ location\_model \blacktriangleright u \mapsto lobby$
            $ownership\_model \cong \texttt{old}\ ownership\_model$
        end
end
```

(b) Eiffel notation

```
class CHAT_SERVER feature

    location_model: ML_MAP[USER,ROOM]
    ownership_model: ML_MAP[ROOM,USER]

    lobby: CHAT_ROOM
    admin: CHAT_USER

    connect (u: USER) is
        require
            a_user /= Void
            not location_model.domain.has_key(u)
        ensure
            location_model |=| old location_model ^ [u, lobby]
            ownership_model |=| old ownership_model
        end
end
```

Figure 4.7: Complete contracts for `connect` via maps

model equal provided they have the same elements in their respective domains and map the same elements in the domain to the associated elements in their respective ranges.

The BON mathematical notation as in (4.1) is often convenient to use. The equivalent Eiffel notation has been designed so as to be as close to the mathematical notation as possible. As shown in Fig. 4.7(b) the Eiffel equivalent of (4.1) is

```
location_model |=| old location_model ^ [u, lobby]
```

The ML library has mathematical maps, sets, bags and sequences and the normal operators of set theory and predicate logic have been implemented (see Chapter 2). For example, the postcondition of query has_user in Fig. 4.10 is specified as

$$Result \triangleq \exists u \in location\_model.domain \bullet (u.user\_name \sim a\_name) \qquad (4.2)$$

The symbol "$\triangleq$" denotes equality by definition. The symbol "$\sim$" denotes object equality (as opposed to reference equality which is denoted by "$=$"). In Eiffel, the effect of $\sim$ is described by the user defined query is_equal. Since the type of *u.user* is STRING, the effect of is_equal for STRING (two strings are equal in this sense if they have the same sequence of characters).

The above postcondition thus asserts that the query holds when there exists some connected user whose name (as a string) has the same characters as the query argument a_name.

An implementor of class `CHAT_SERVER` may provide any private implementation code that satisfies the ML-Contracts. For example, the implementor may use two linked lists (`users` and `rooms`) for the implementation. However, all the contracts are specified in terms of the model. Thus, the implementor must link the implementation to the model by providing an abstraction function [49] that maps (or "lifts") the implementation detail to the model as shown in Fig. 4.10. For example for the location model, the abstraction function is

$$Result \equiv \langle\langle i : INT | 0 \leq i < users.count \bullet users[i] \mapsto users[i].room \rangle\rangle \qquad (4.3)$$

The angle brackets $\langle\langle \cdots \rangle\rangle$ is used for map comprehension (similar to set comprehension). Thus, (4.3) asserts that the location model is a map consisting of pairs $users[i] \mapsto users[i].room$ where $users[i]$ is the item (i.e., the user) at index i in the linked list *users*.

In addition to the location model, we will also need an ownership model which is a map from rooms to users (owners) as shown in Fig. 4.9. For example, when a user adds a new room it is the ownership model that changes while the location model remains the same (e.g., see routine `add_room` in Fig. 4.5). The domain of the ownership model is the set of all rooms in the chat application. The contracts (expressed in terms of the model) for the chat application are shown in more detail in Fig. 4.5.

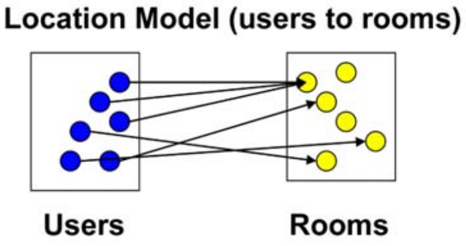


Figure 4.8: Location model (mapping from users to rooms)

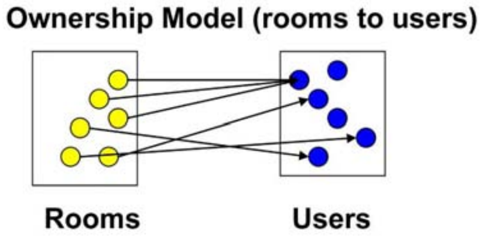
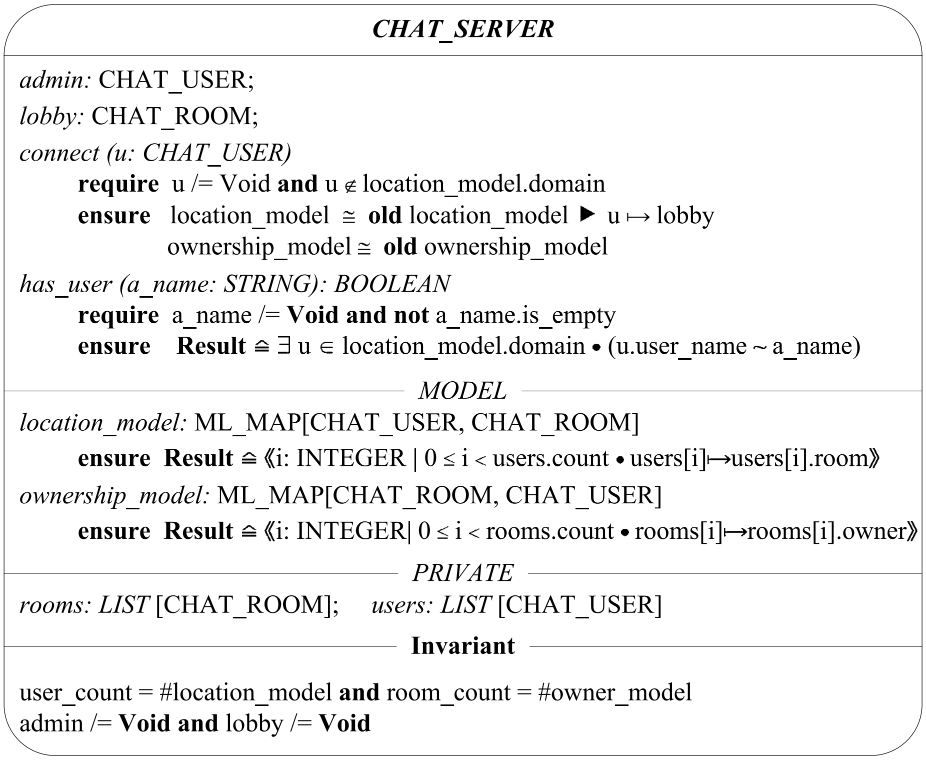


Figure 4.9: Ownership model (mapping from rooms to users)



```
CHAT_SERVER
admin: CHAT_USER;
lobby: CHAT_ROOM;
connect (u: CHAT_USER)
        require  u /= Void and u ∉ location_model.domain
        ensure   location_model ≅ old location_model ▶ u ↦ lobby
                 ownership_model ≅ old ownership_model
has_user (a_name: STRING): BOOLEAN
        require  a_name /= Void and not a_name.is_empty
        ensure   Result ≙ ∃ u ∈ location_model.domain • (u.user_name ~ a_name)
                                   MODEL
location_model: ML_MAP[CHAT_USER, CHAT_ROOM]
        ensure  Result ≙《i: INTEGER | 0 ≤ i < users.count • users[i]↦users[i].room》
ownership_model: ML_MAP[CHAT_ROOM, CHAT_USER]
        ensure  Result ≙《i: INTEGER| 0 ≤ i < rooms.count • rooms[i]↦rooms[i].owner》
                                   PRIVATE
rooms: LIST [CHAT_ROOM];    users: LIST [CHAT_USER]
                                   Invariant
user_count = #location_model and room_count = #owner_model
admin /= Void and lobby /= Void
```

Figure 4.10: BON specification of the chat server

4.6 Contract violations in Fit tables

Section 4.4 used Scenario Tests to specify the design. Fit tables were able to catch specification errors (as category 1 errors in which the expected value disagreed with the actual values) in the design (category 1 errors may also reflect implementation errors). Section 4.5 used ML-Contracts to specify the design. The advantage of ML-Contracts (as opposed to Scenario Tests for a particular execution) is that they specify the complete behaviour of modules (classes). In ESpec, such contract violations are reflected back into the Fit tables (these are category 2 errors). This is useful because a contract error in the Fit table indicates that the specification of the design is correct, but the implementation does not satisfy the specified design solution. The contract violation in the Fit table provides precise details as to which feature fails which makes it easier to fix the problem.

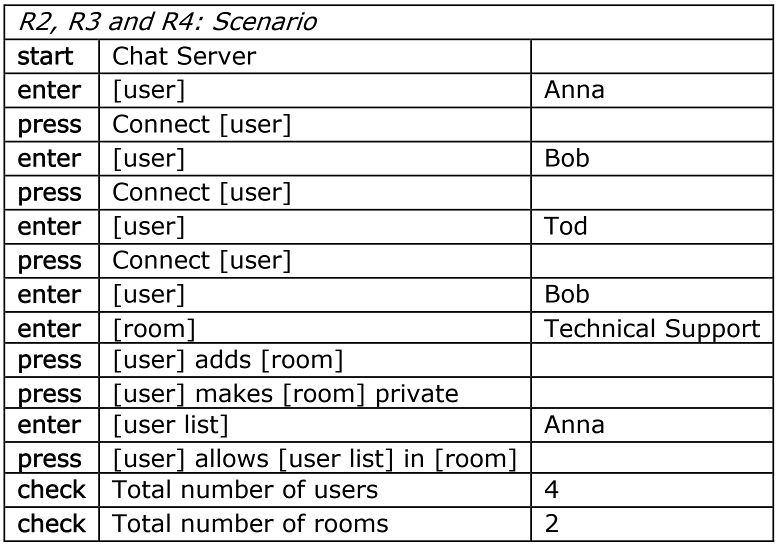
We illustrate the use of contract violations in Fit tables with some new Fit tables in our requirement document as shown in Fig. 4.11 and Fig. 4.12. These tables convert requirements R2, R3, and R4 into a mechanically testable format.

We use an Action Table to specify a sequence of actions such as adding users, rooms and permissions. We then use a Row Table to query and check that the underlying database of users, rooms and permissions are as expected.

Row Tables allow for powerful descriptions that collections of elements (e.g.,

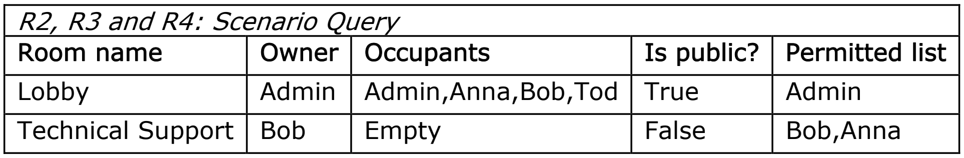
Action Table:

- Start a chat server, create three chat users (“Anna”, “Bob” and “Tod”) and connect them to the server.
- User “Bob” creates a chat room called “Technical Support” and adds it to the chat server.
- “Bob” changes the room status from public to private.
- “Bob” permits user “Anna” to join the room.



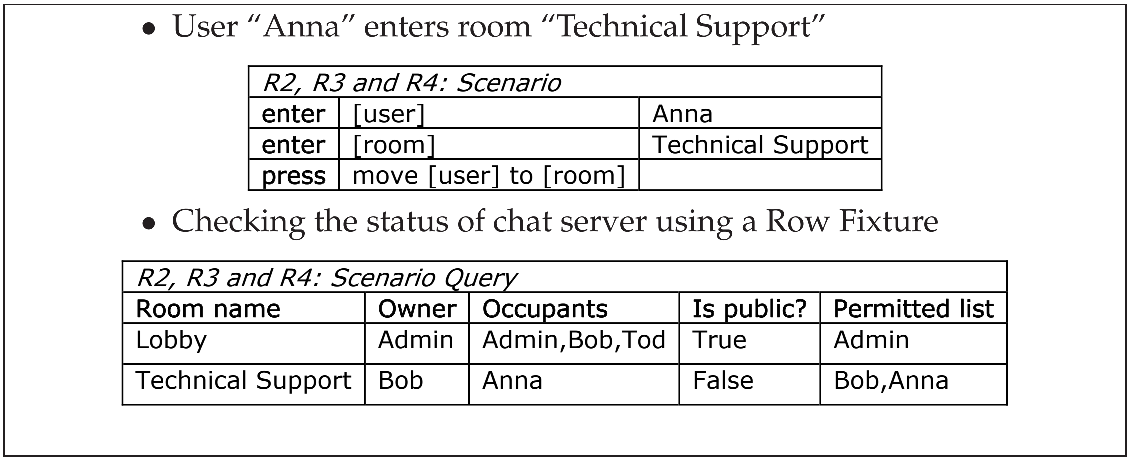
R2, R3 and R4: Scenario		
start	Chat Server	
enter	[user]	Anna
press	Connect [user]	
enter	[user]	Bob
press	Connect [user]	
enter	[user]	Tod
press	Connect [user]	
enter	[user]	Bob
enter	[room]	Technical Support
press	[user] adds [room]	
press	[user] makes [room] private	
enter	[user list]	Anna
press	[user] allows [user list] in [room]	
check	Total number of users	4
check	Total number of rooms	2

Row Table:
The following Row table checks the status of the database of users and rooms.



R2, R3 and R4: Scenario Query				
Room name	**Owner**	**Occupants**	**Is public?**	**Permitted list**
Lobby	Admin	Admin,Anna,Bob,Tod	True	Admin
Technical Support	Bob	Empty	False	Bob,Anna

Figure 4.11: Testing requirements R2—R4



- User “Anna” enters room “Technical Support”

R2, R3 and R4: Scenario		
enter	[user]	Anna
enter	[room]	Technical Support
press	move [user] to [room]	

- Checking the status of chat server using a Row Fixture

R2, R3 and R4: Scenario Query				
Room name	**Owner**	**Occupants**	**Is public?**	**Permitted list**
Lobby	Admin	Admin,Bob,Tod	True	Admin
Technical Support	Bob	Anna	False	Bob,Anna

Figure 4.12: Moving a user from one room to another

in lists, sets, bags and maps) are present as expected. For example, suppose only “Bob” and “Anna” have been allowed to access the room “Technical Support”. A single row in a Row Table can check that these users alone are in the permitted list by simple enumeration.

Fig. 4.11 has an Action Table and a Row Table. The Action Table starts a chat server, creates three chat users (“Anna”, “Bob”, and “Tod”) and connects them to the server (“Connect [user]” at rows 4, 6, and 8). Then, “Bob” creates a chat room called “Technical Support” and adds it to the chat server (“[user] adds [room]” at row 11). Next, “Bob” changes the room status to private (“[user] makes [room] private” at row 12) and permits “Anna” to join the room (“[user] allows [user list] in [room]” at row 14). At this point, we expect that there must be two rooms and four users (including the administrator) in the chat server. This is checked by “Total number of users” at line 14 and “Total number of rooms” at row 16.

Consider the Row Table in Fig. 4.11. The customer specifies the header of the first column in the table as “Room name”. This means that the customer will be querying a collection of entities of type room considered as phenomena in the problem space.

Each row in the Row Table describes the properties of a room in the collection. In the Row Table of Fig. 4.11 the first row deals with room “Lobby” and the second row deals with room “Technical Support”. These are the only two rows because the customer has not created any other rooms. Suppose there are other rooms but they are not expected in the table. Then execution of the Row Table would yield an error stating that there are surplus rooms in the business logic that were not expected in the requirements. Thus, Row tables can represent *exhaustive* descriptions of the collection.

The other column headings of the Row Table in Fig. 4.11 describe properties that each room must satisfy. The second column, for example, specifies who is the owner of the room, the third column describes who are the occupants of the room, the fourth column asserts whether the room is public (anybody may enter), and the last column checks the permitted list. If the room is public then the permitted list contains only the owner of the room.

It is of course up to the developer to connect the Row Table to the business logic via a Row Fixture (`CHAT_ROW`). The Fixture code associated with the Row

Table (4.11) is shown in Section B.2.

Row and Action Tables in Fig. 4.11 are connected to each other. This is achievable via using a construct called `connect_row_to_action`. This allows a Row Fixture to access objects that are initialized in an Action Fixture. For example, in the `CHAT_ROW` Fixture, we would like to access the list of rooms and list of users of the `server` object which was initially created by the `CHAT_ACTION` Fixture.

The connection is made in the root class which inherits from `ES_SUITE`. The root class is shown in listing 4.13. The root class inherits from `ES_SUITE`. In the root class we create the Scenario Tests and the Fixture objects. We also connect the Fit tables to the corresponding Fixture objects. For example, we've bound the name of the tables "R2, R3 and R4: Scenario" and "R2, R3 and R4: Scenario Query" to the corresponding classes `CHAT_ACTION` and `CHAT_ROW` via the `add_Fixture` construct (lines 11–13 of Fig. 4.13). The connection between the Row Fixture and the Action Fixture is made at line 16.

Tables in Fig. 4.12 are continuation of the requirement document described in Fig. 4.11. Since the tables have the same title they refer to the same chat server initialized and acted upon in Fig. 4.11. Subsequent to the actions of Fig. 4.11, our customer uses the Action Table in Fig. 4.12 to specify that user "Anna" moves from the "Lobby" to "Technical Support". As shown in the Row Table, our customer expects that user "Anna" is transferred from the "Lobby" to "Technical

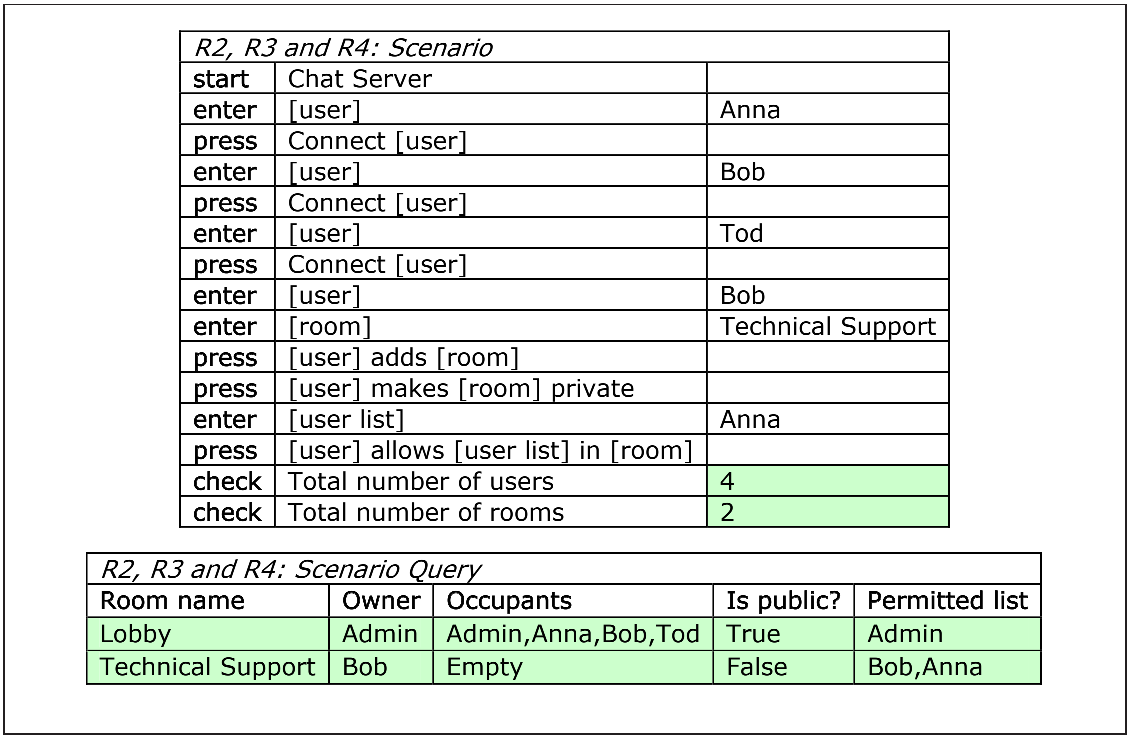
```
class ROOT_CLASS inherit
  ES_SUITE
create
    make

feature -- Create

make is
  do
    -- Binding Fixture to the corresponding tables in the HTML document
    add_fixture("R1: Chat Server Setup", create {CHAT_ACTION}.make)
    add_fixture("R2, R3 and R4: Scenario", create {CHAT_ACTION}.make)
    add_fixture("R2, R3 and R4: Scenario Query", create {CHAT_ROW}.make)

    -- Connecting Row to Action Tables
    connect_row_to_action("R2, R3 and R4: Scenario", "R2, R3 and R4: Scenario Query")

    -- Including Scenario Tests
    add_test (create {CHAT_TEST1}.make)

    -- Show Contract Violations
    show_errors

    -- Execute ESpec
    run_espec
  end

end -- class ROOT_CLASS
```

Figure 4.13: The root class for testing our system

Support".

Do the Fit tests pass given the design developed in previous section? If we execute the Fit requirement tests described in Fig. 4.11 and Fig. 4.12 we obtain the results shown in Fig. 4.14 and Fig. 4.15.

Both tables in Fig. 4.14 succeed whereas the Row Table in Fig 4.15 fails with a category 1 error indicating that "Anna" is in two locations at the same time



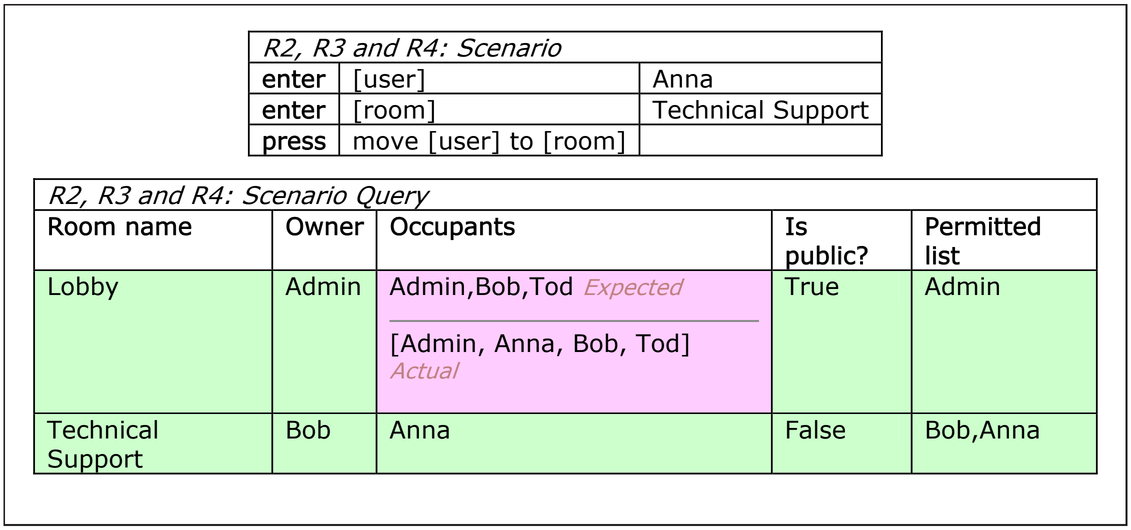
R2, R3 and R4: Scenario		
start	Chat Server	
enter	[user]	Anna
press	Connect [user]	
enter	[user]	Bob
press	Connect [user]	
enter	[user]	Tod
press	Connect [user]	
enter	[user]	Bob
enter	[room]	Technical Support
press	[user] adds [room]	
press	[user] makes [room] private	
enter	[user list]	Anna
press	[user] allows [user list] in [room]	
check	Total number of users	4
check	Total number of rooms	2

R2, R3 and R4: Scenario Query				
Room name	**Owner**	**Occupants**	**Is public?**	**Permitted list**
Lobby	Admin	Admin,Anna,Bob,Tod	True	Admin
Technical Support	Bob	Empty	False	Bob,Anna

Figure 4.14: Success: Result of executing tables in Fig. 4.11

(in the "Lobby" and "Technical Support"). According to the Row Table, after moving from one room to another, the customer's expectation is that "Anna" is solely in "Technical Support" and not in the "Lobby" anymore.

An investigation of the code shows an implementation error in the body of routine `CHAT_SERVER.enter_room` (see Section B.7 for `CHAT_SERVER` code). The developer simply forgot to remove the user from the original room while adding this user to the new room thus causing the user to be in two locations at the same time. This category 1 error in the Fit table is an indication of an incomplete specification as the error should have been caught by a contract violation (i.e., a



R2, R3 and R4: Scenario		
enter	[user]	Anna
enter	[room]	Technical Support
press	move [user] to [room]	

R2, R3 and R4: Scenario Query				
Room name	Owner	Occupants	Is public?	Permitted list
Lobby	Admin	Admin,Bob,Tod *Expected* [Admin, Anna, Bob, Tod] *Actual*	True	Admin
Technical Support	Bob	Anna	False	Bob,Anna

Figure 4.15: Failure: Result of executing tables in Fig 4.12

category 2 error).

The fix for this problem is to convert a category 1 error into a category 2 contract error. Consider the specification of routine `enter_room` in Fig. 4.16. The postcondition is:

$$location\_model \quad \cong \quad (\textbf{old}\ location\_model) \oplus (u \mapsto get\_room(r))$$

where $\oplus$ is the symbol for map override. The postcondition asserts that the location model in the poststate is the same as in the prestate except that the room associated with user u is now changed to $get\_room(r)$ where query `get_room` returns the chat room object associated with string r (this is a search routine). The ownership model is left unchanged by the routine `enter_room`.

```
class CHAT_SERVER
    ...
    enter_room (u: CHAT_USER; r: STRING) is
            -- Move user 'u' into room with string name 'r'
        require
            $(u \neq Void) \wedge (r \neq Void) \wedge \neg(r.is\_empty)$
            $u \in location\_model.domain \wedge has\_room(r)$
        ensure
            user_entered: $location\_model \cong (\texttt{old}\ location\_model \oplus (u \mapsto get\_room(r))$
            ownerships_not_changed: $ownership\_model \cong \texttt{old}\ ownership\_model$
        end

    get_room (r: STRING): CHAT_ROOM is
            -- returns a room with name 'r'
        require
            $(r \neq Void) \wedge \neg(r.is\_empty)$
            has_room (r)
        ensure
            $(\exists\ room \in ownership\_model.domain \mid room.name \sim Result.name)$
        end

    ...
    location_model: ML_MAP [CHAT_USER, CHAT_ROOM]
    ownership_model: ML_MAP [CHAT_ROOM, CHAT_USER]

invariant
    disjoint_users:
    $(\forall\ r_1, r_2 \in own\_model.domain \mid r_1 \neq r_2\ \bullet r_1.occupant\_model\ \cap\ r_2.occupant\_model \cong \emptyset)$
    coverage: $(\cup\, r \in own\_model.domain \bullet r.occupant\_model) \cong location\_model.domain$
end
```

Figure 4.16: BON specification of the invariant for the CHAT_SERVER

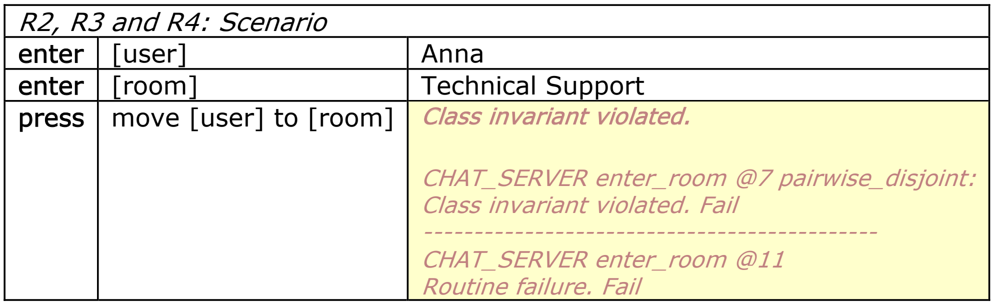
The postcondition is correct but incomplete (as the error was category 1 and not category 2). As shown in Fig. 4.5, class CHAT_ROOM has an *occupant model* (ML_SET [CHAT_USER]) that keeps track of the occupants of each room. There is no assertion that ensures that the models of CHAT_SERVER and CHAT_ROOM are consistent with each other. The consistency assertions are best written as invariants in

```
class CHAT_SERVER
   ...
    location_model: ML_MAP [USER, ROOM]
    ownership_model: ML_MAP [ROOM, USER]

    forall_rooms (r1: CHAT_ROOM): BOOLEAN is
        do
            Result := ownership_model.domain.for_all
                        (agent empty_intersection (r1, ?))
        end

    empty_intersection (r1, r2: CHAT_ROOM): BOOLEAN is
        do
            if r1 /= r2 then
                Result := (r1.occupant_model * r2.occupant_model).is_empty
            else
                Result := true
            end
        end

    multi_union (s: ML_SET[CHAT_ROOM]): ML_SET[CHAT_USER] is
        local
            seq: ML_SEQ[CHAT_ROOM]
        do
            seq := s.to_seq
            if seq.count = 1 then
                Result := seq.head.occupant_model.to_set
            else
                Result := (multi_union (seq.tail.to_set) |++
                                (seq.head.occupant_model)).to_set
            end
        end

invariant
    pairwise_disjoint:
        ownership_model.domain.for_all (agent forall_rooms (?))
    coverage:
        multi_union (ownership_model.domain) |=| location_model.domain
```

Figure 4.17: Implementing the invariant using ML

class CHAT_SERVER as shown in Fig. 4.16.

The invariant disjoint_users asserts that any two rooms are pairwise disjoint, i.e., a user may be in at most one room at a time. The second invariant `coverage` asserts that the domain of the location model (i.e., all chat users) consists of the union of the occupant model sets, i.e., all users specified in the location model must be occupants of some room.



R2, R3 and R4: Scenario		
enter	[user]	Anna
enter	[room]	Technical Support
press	move [user] to [room]	*Class invariant violated.* *CHAT_SERVER enter_room @7 pairwise_disjoint:* *Class invariant violated. Fail* --- *CHAT_SERVER enter_room @11* *Routine failure. Fail*

Table 4.4: Specification violations are reflected to the Fit table

Fig. 4.17 shows how the invariants are written using the ML-Contracts. The invariant `disjoint_users` is captured by enumerating through the list of all rooms collecting, pairwise, intersections of the room occupants and then checking that the resulting set is empty. Queries `forall_rooms` and `empty_intersection` are agent routines that are used for this purpose (see lines 6–19 in Fig. 4.17). The "`*`" infix operator is used for the intersection of two sets. The `coverage` property is implemented using the `multi_union` recursive agent. This agent collects the union of all users in all rooms using the `occupant_model` of each room and then returns a set composed of all those users. The "`|++`" infix operator is used for the union of two sets.

The Fit table now reports an invariant error in class CHAT_SERVER (i.e., a category 2 error) thus indicating an implementation problem in routine enter_room in class CHAT_SERVER. This contract error is reported in Table 4.4.

4.7 Security issues

Naturally, security is one of the most important aspects that we have to keep in mind when developing a chat application. We can think of security at any level of abstraction. Customers can assert the security rules that are important to them in the Fit tables at the level of the problem domain. In the solution space, the developers can write ML-Contracts or Scenario Tests for checking the security rules. Violation test cases can be written for checking that contracts correctly capture those rules. The following is an example of a security rule:

> **Security Rule:** a chat user not in the permitted list of a private room should not be allowed to enter the room.

In terms of the actual design, we use the Eiffel export declaration construct to impose the security constraint. In the BON diagram Fig. 4.3, the feature users in class CHAT_SERVER is not exported to a CHAT_USER. Whatever our design, we need to test that the requirement has been satisfied.

In order to assert the above security rule, our customer creates a series of Fit tables. This is shown in Figures 4.5– 4.7.

- Connect 4 users to the server followed by adding 2 rooms

Chat Security		
start	Chat Server	
enter	[user list]	Anna,Bob,Jackie,Tod
press	Connect [user list]	
enter	[user]	Anna
enter	[room]	Gold service
press	[user] adds [room]	
enter	[user]	Bob
enter	[room]	Silver service
press	[user] adds [room]	

- Check rooms contents

Chat Security Query				
Room name	**Owner**	**Allowed list**	**Is private?**	**Occupants**
Gold service	Anna	N/A	False	empty
Silver service	Bob	N/A	False	empty
Lobby	Admin	N/A	False	Admin,Anna,Bob,Jackie,Tod

Table 4.5: Setting up the server, users and rooms

- "Anna", "Jackie" and "Tod" move to room "Gold service"

Chat Security		
enter	[user]	Anna
enter	[room]	Gold service
press	[user] enters [room]	
enter	[user]	Jackie
press	[user] enters [room]	
enter	[user]	Tod
press	[user] enters [room]	

- Checking rooms contents after the move

Chat Security Query				
Room name	**Owner**	**Allowed list**	**Is private?**	**Occupants**
Gold service	Anna	N/A	False	Anna,Jackie,Tod
Silver service	Bob	N/A	False	empty
Lobby	Admin	N/A	False	Admin,Bob

Table 4.6: Moving users

- "Anna" changes the status of her room ("Gold service") to private and allows "Tod" and "Bob" to access the room

Chat Security		
enter	[user]	Anna
enter	[room]	Gold service
enter	[user list]	Tod,Bob
press	[user] allows [user list] to access [room]	
press	[user] sets [room] to private	
check	Is [room] private?	True

- Checking rooms contents

Chat Security Query				
Room name	Owner	Allowed list	Is private?	Occupants
Gold service	Anna	Anna,Tod,Bob	True	Anna,Tod
Silver service	Bob	N/A	False	empty
Lobby	Admin	N/A	False	Admin,Bob,Jackie

Table 4.7: Changing room status to private allowing users to access

The first table in Fig. 4.5 starts a chat server and connects four users ("Anna", "Bob", "Jackie" and "Tod") to the server. Users "Anna" and "Bob" add two rooms to the server ("Gold service" and "Silver service" respectively). The following Row Table checks the rooms of the server by examining various properties about the rooms. If the rooms are public then anybody may join and thus the allowed list constraint is marked as "N/A" (not applicable).

In the first table of Fig. 4.6, we move three users ("Anna", "Jackie" and "Tod") to the "Gold service" room. Since "Gold service" is a public room at this point, we expect that all three users enter it without any problems. This is tested in the next table of Fig. 4.6 which checks the contents of each room to make sure that the three users have successfully moved to the "Gold service" room.

In Fig. 4.7, “Anna” adds two users (“Bob” and “Tod”) to the allowed list of people who can enter her room. This is followed by changing the room status to private. The second table of Fig. 4.7 explicitly checks that the above actions are done correctly, i.e., *allowed list* has been updated and the room has become private.

The concrete example described in the above tables, can be used by the developer in order to come up with the right specification for the code. We can focus on the development of the `set_private` feature. This feature will be called by a user and its job is to set room’s status to private. According to the Fit table, calling this method not only should change the room’s status (from public to private) but also should force all un-authorized users out of the room to the lobby. The design is shown in Figures 4.18 and 4.19.

```
set_private (u: CHAT_USER; room: STRING) is
        -- sets the room to private
    require
        $u \neq \texttt{void} \wedge room \neq \texttt{void} \wedge \neg room.is\_empty$
        $has\_room(room)$
        $has\_user(u.user\_name)$
        $get\_room(room).owner = u$
    ensure
        $get\_room(name).is\_private$
        $\forall\ (u \mapsto r) \in \texttt{old}\ location\_model\ \bullet$
                        $(is\_allowed(u, r.name) \rightarrow (u \mapsto r) \in\ location\_model) \wedge$
                        $\neg is\_allowed(u, r.name) \rightarrow (u \mapsto lobby) \in\ location\_model)$
        $owership\_model \cong \texttt{old}\ owership\_model$
```

Figure 4.18: `CHAT_SERVER.set_private`

```
    occupant_model: ML_SET [CHAT_USER]
        ...
    set_private is
            -- set room to be of type private
        require
            ¬is_private
        ensure
            is_private
        ...
invariant
    no_unauthorized_users: ∀ u ∈ occupant_model • is_allowed(u)
```

Figure 4.19: `CHAT_ROOM.set_private`

The specification for `set_private` is as follows: (a) the room status must change to private, (b) all users who are not allowed, should be moved to the lobby room and all users who are allowed, remain in their original rooms and (c) ownership model remains unchanged.

As we mentioned in Chapter 2, it is not easy to describe assertions such as (b) using basic Eiffel contracting mechanism. However, these assertions can be easily described in the terms of our models. The invariant of `CHAT_ROOM` class (see Fig. 4.19) asserts that all users in a room must be authorized.

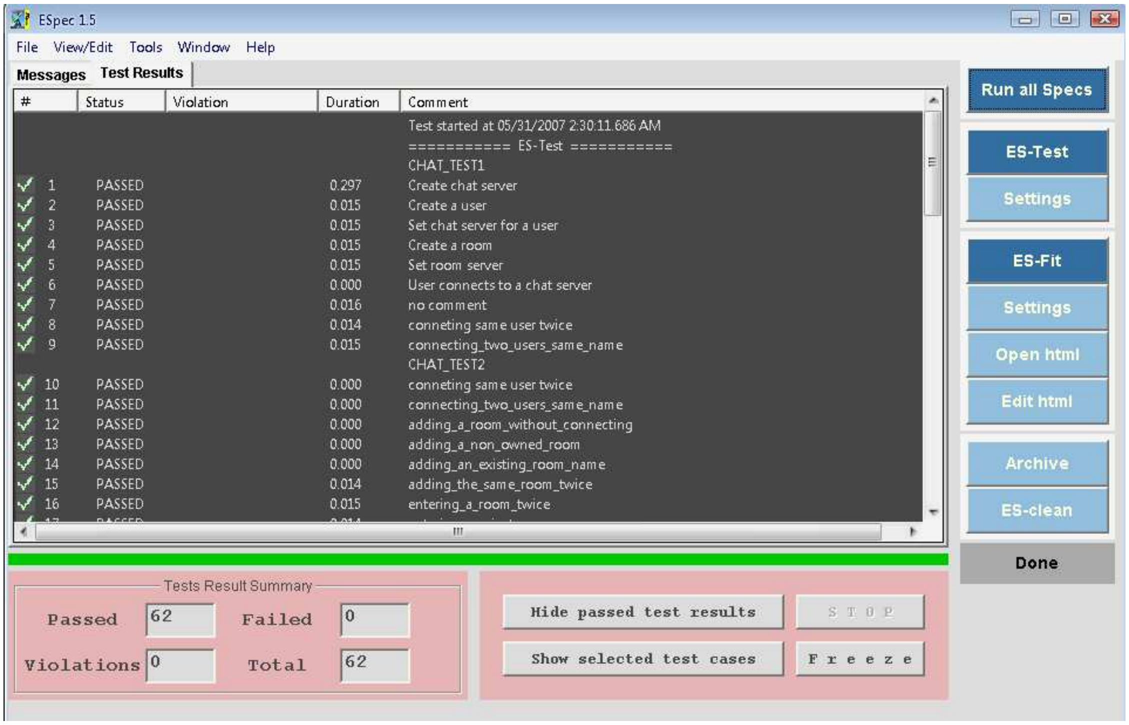
4.8 Conclusions

In this chapter, we showed snippets of the development process of a chat application using our method.

Complete code, contracts and tests for this example are provided in Appendix B. We used the ESpec tool in order to capture the customer's informal requirements in the form of testable Fit tables and then we used these tables to drive the design. During the design phase we used ML-Contracts and Scenario Tests to write and test the design specifications for the various modules.

The ESpec tool tests the requirements (Fit tables) and specifications (ML-Contracts and Scenario Tests) under the control of a single green bar as shown in Fig. 4.8. If all the tests pass, then we have checked the design and validated that the design satisfies the customer requirements.

The chat room example thus illustrates our method of early testable requirements and specifications which shows that our approach and tool is an integrated wide-spectrum (i.e., deals both with customer requirements and design specifications) method.



#	Status	Violation	Duration	Comment
				Test started at 05/31/2007 2:30:11.686 AM
				=========== ES-Test ===========
				CHAT_TEST1
✓ 1	PASSED		0.297	Create chat server
✓ 2	PASSED		0.015	Create a user
✓ 3	PASSED		0.015	Set chat server for a user
✓ 4	PASSED		0.015	Create a room
✓ 5	PASSED		0.015	Set room server
✓ 6	PASSED		0.000	User connects to a chat server
✓ 7	PASSED		0.016	no comment
✓ 8	PASSED		0.014	conneting same user twice
✓ 9	PASSED		0.015	connecting_two_users_same_name
				CHAT_TEST2
✓ 10	PASSED		0.000	conneting same user twice
✓ 11	PASSED		0.000	connecting_two_users_same_name
✓ 12	PASSED		0.000	adding_a_room_without_connecting
✓ 13	PASSED		0.000	adding_a_non_owned_room
✓ 14	PASSED		0.000	adding_an_existing_room_name
✓ 15	PASSED		0.014	adding_the_same_room_twice
✓ 16	PASSED		0.015	entering_a_room_twice

Tests Result Summary

Passed 62 Failed 0

Violations 0 Total 62

Table 4.8: Snapshot of ESpec after running all the tests

5 Design of the ESpec Tool

In previous chapters we illustrated our method of Testable Specifications and Requirements using Eiffel and ESpec. This chapter describes the design and implementation of the ESpec tool itself. In the following sections we'll describe the challenges that we faced in developing various components of our tool and our design decisions.

ESpec has two major components as shown in the UML deployment diagram in Fig. 5.1:

1. **ESpec Library:** which contains the core components of the tool. This library must be included as part of the system under test. It provides enough facilities for the developers to use ESpec at the command line or Eiffel Studio's integrated development environment.

2. **ESpec GUI:** which is the graphical user interface component of the tool. The ESpec GUI communicates with the ESpec library over a socket connection making it completely independent of the business logic and the

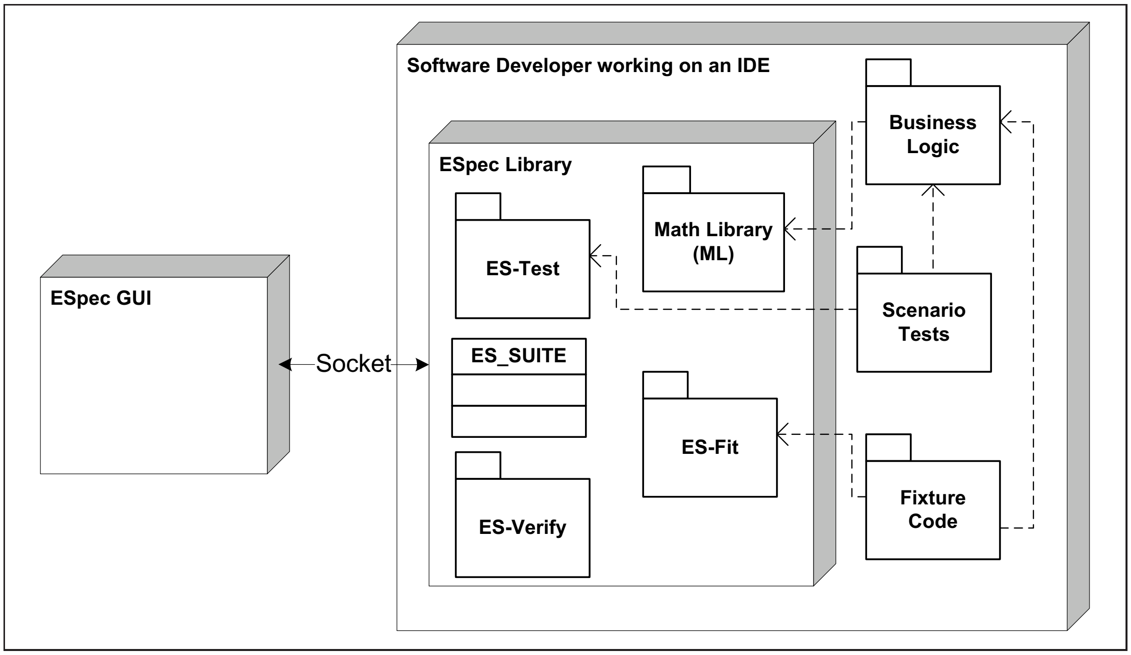


Figure 5.1: Deployment diagram of the components of the ESpec tool

system under test. The GUI helps the developers work at a faster pace and provides a familiar way for the customers to develop and test the software product.

Consider software developers who would like to test their business logic. The tests (e.g. Fit and Scenario tests) will exercise the business logic and report results back to the ESpec GUI. Test results should be reported in real-time to the GUI as they execute. How will the tests and the GUI communicate with each other. Our solution is as follows. The developer imports the ESpec libraries which communicate with the GUI via sockets. The tests may also be run from the command line in which case the sockets are not used.

The library is divided into four main subcomponents: (1) ES-Fit, (2) ES-Test, (3) ES-Verify, and (4) ML—Mathematical Library. These components are connected to each other using a shared interface which we call `ES_SUITE`. `ES_SUITE` allows the developers to access any of the above tools individually or in combination with other tools. But before discussing the design of `ES_SUITE`, we need to understand each of the above components. The next section is devoted to ES-Fit tool and the challenges associated with implementing it in Eiffel language.

5.1 ES-Fit Architecture

Classes associated with ES-Fit are located in the ES-FIT subcluster in the ESPEC library. ES-Fit goes through five main steps to read and execute the Fit tables. These essential steps are shown in Fig. 5.2: (1) Parsing the input HTML file; (2) Running the Fixture code associated with each table; (3) Converting the type of strings read from the tables to the appropriate types; (4) Executing the Fixture code and compare the results to the expected values from the tables; (5) Reporting the results back to the HTML document. In the following sections, we will describe various ES-Fit modules that handle each step.

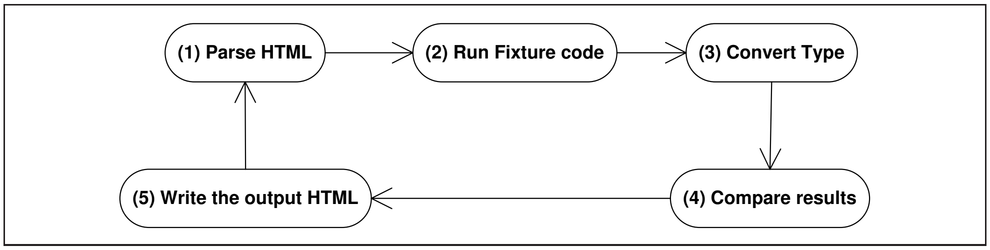


Figure 5.2: Activity diagram of Fit processing steps

5.1.1 HTML parser

The parser handles any HTML document. The advantage of representing our requirement description documents as HTML is that it allows customers to use their standard document preparation tools (e.g., Word, Open Office, Excel) because all these tools generate HTML.

ES-Fit reads the input HTML document and then parses it into tokens. The parser builds an internal model of the HTML document preserving all of the content while exposing the Fit table data to further processing.

The visible text within a cell of each table is extracted and treated as a string, free of formatting. Markup tags, character escapes, and leading and trailing spaces are all removed. The parsed tables can be modified in memory and a revised document can be written with feedback for the user. The revised document is an extended version of the input document which includes cell background colors, cell contents, additional rows or additional columns. The parser looks for one tag at a time. A matching pair, including both start and end, e.g., $< table >$ and $< /table >$, are stored in a single parse node. This same node stores the text before, between and after the tags. These are called the *leader*, *body* and *trailer*, respectively, and are of type STRING.

Should the trailer contain another tag then it is parsed and the result stored in attribute *more*. Likewise, if the body contains additional tags then these are parsed and the result stored in attribute *parts*. The result is a parse tree which happens to be a binary tree where *parts* is the left subtree and more is the *right* subtree. A depth-first traversal visits each cell in the natural reading order of top down, left to right.

In ES-Fit, the parsing is done in class ES_HTML_PARSER. This class follows the Fit specification, therefore any HTML document that follows the Fit standard is supported by the parser. A sequence of tables in a file and the subcomponents of those tables are represented by objects of class ES_HTML_PARSER. As an example, the parse structure of Table 5.1 is illustrated in Fig. 5.3.

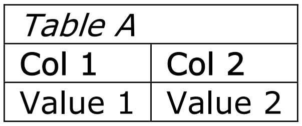


Table A	
Col 1	Col 2
Value 1	Value 2

Table 5.1: Sample Fit table

In Fig. 5.3 we show instances (objects) of class ES_HTML_PARSER that represent

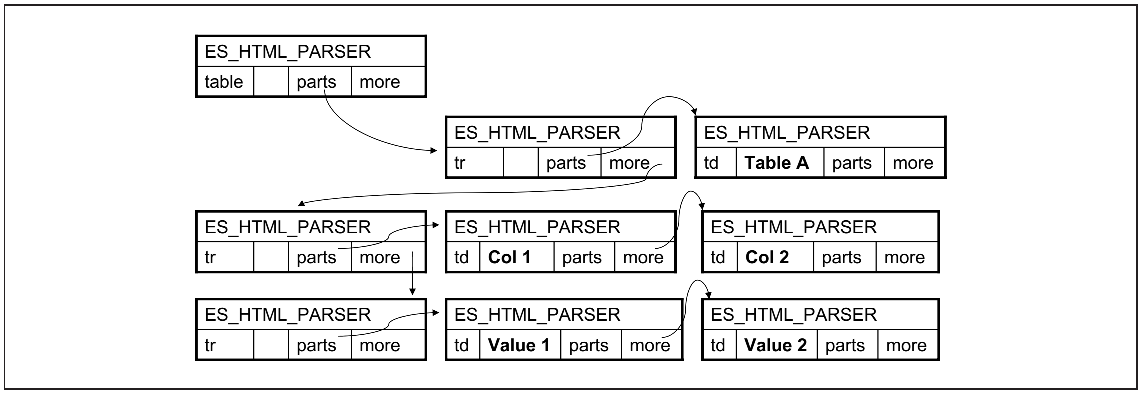
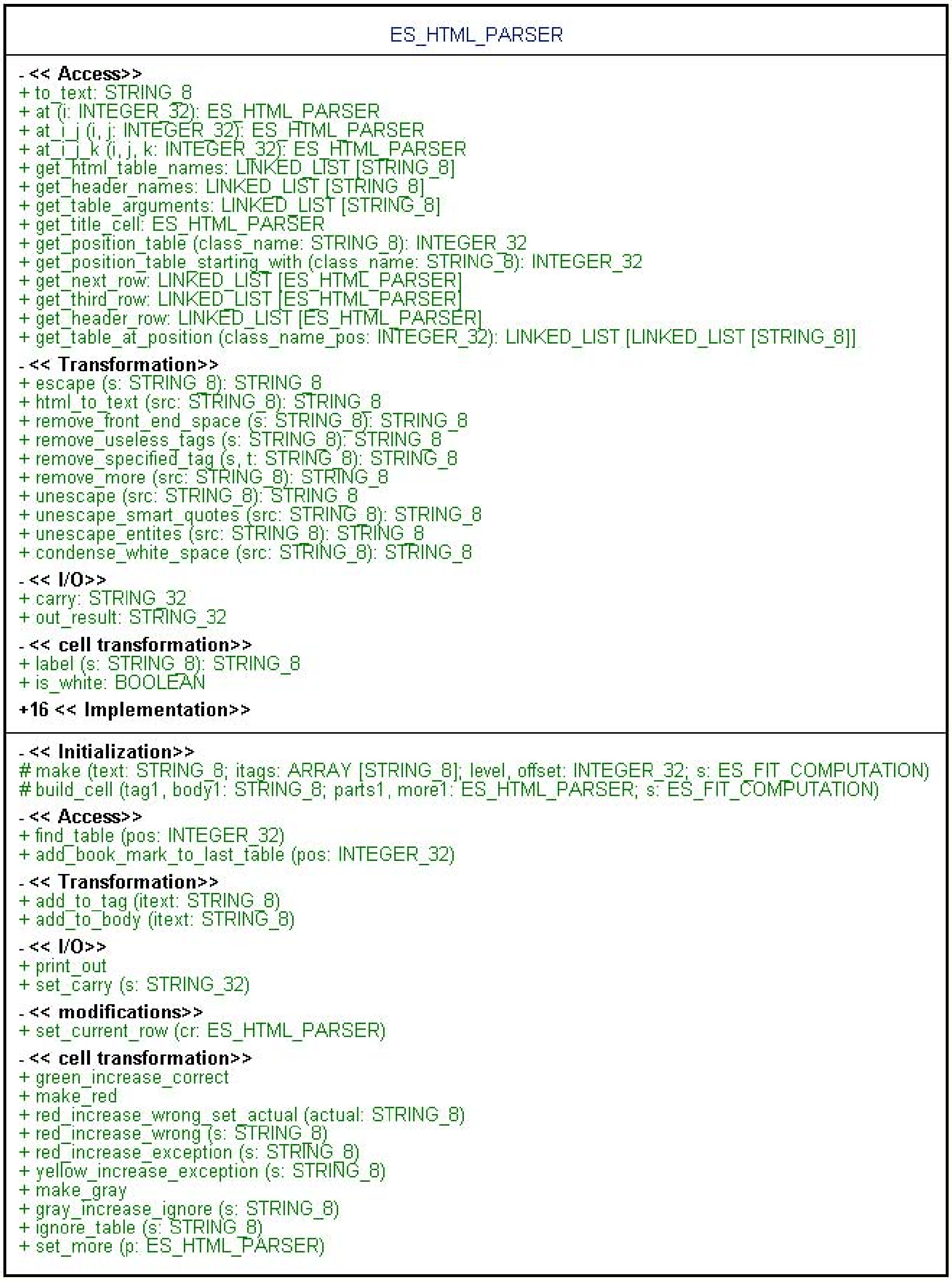


Figure 5.3: Parse structure for table 5.1

the indicated Fit table. Each object is shown with four fields: *type*, *body*, *parts* and *more*. The figure shows the values of each of these fields in each object.

The field *type* in each object takes one of three possible values: *table*, *tr* (representing a table row) or *td* (represent a table cell), corresponding to the HTML tags. The *body* contains the string representation of the text inside the cell (when object is of type *td*).

The top ES_HTML_PARSE object represents the whole table, parts refer to a sequence of *tr* ES_HTML_PARSE objects that represent the rows of the table; *more* refers to the next table, if any. Each *tr* (row) object in turn refers to a sequence of *td* (column) objects. The sequence of components is formed by following the *more* values. Fig. 5.4 shows the list of available features in the ES_HTML_PARSE class.



```
ES_HTML_PARSER

- << Access>>
+ to_text: STRING_8
+ at (i: INTEGER_32): ES_HTML_PARSER
+ at_i_j (i, j: INTEGER_32): ES_HTML_PARSER
+ at_i_j_k (i, j, k: INTEGER_32): ES_HTML_PARSER
+ get_html_table_names: LINKED_LIST [STRING_8]
+ get_header_names: LINKED_LIST [STRING_8]
+ get_table_arguments: LINKED_LIST [STRING_8]
+ get_title_cell: ES_HTML_PARSER
+ get_position_table (class_name: STRING_8): INTEGER_32
+ get_position_table_starting_with (class_name: STRING_8): INTEGER_32
+ get_next_row: LINKED_LIST [ES_HTML_PARSER]
+ get_third_row: LINKED_LIST [ES_HTML_PARSER]
+ get_header_row: LINKED_LIST [ES_HTML_PARSER]
+ get_table_at_position (class_name_pos: INTEGER_32): LINKED_LIST [LINKED_LIST [STRING_8]]
- << Transformation>>
+ escape (s: STRING_8): STRING_8
+ html_to_text (src: STRING_8): STRING_8
+ remove_front_end_space (s: STRING_8): STRING_8
+ remove_useless_tags (s: STRING_8): STRING_8
+ remove_specified_tag (s, t: STRING_8): STRING_8
+ remove_more (src: STRING_8): STRING_8
+ unescape (src: STRING_8): STRING_8
+ unescape_smart_quotes (src: STRING_8): STRING_8
+ unescape_entites (src: STRING_8): STRING_8
+ condense_white_space (src: STRING_8): STRING_8
- << I/O>>
+ carry: STRING_32
+ out_result: STRING_32
- << cell transformation>>
+ label (s: STRING_8): STRING_8
+ is_white: BOOLEAN
+16 << Implementation>>

- << Initialization>>
# make (text: STRING_8; itags: ARRAY [STRING_8]; level, offset: INTEGER_32; s: ES_FIT_COMPUTATION)
# build_cell (tag1, body1: STRING_8; parts1, more1: ES_HTML_PARSER; s: ES_FIT_COMPUTATION)
- << Access>>
+ find_table (pos: INTEGER_32)
+ add_book_mark_to_last_table (pos: INTEGER_32)
- << Transformation>>
+ add_to_tag (itext: STRING_8)
+ add_to_body (itext: STRING_8)
- << I/O>>
+ print_out
+ set_carry (s: STRING_32)
- << modifications>>
+ set_current_row (cr: ES_HTML_PARSER)
- << cell transformation>>
+ green_increase_correct
+ make_red
+ red_increase_wrong_set_actual (actual: STRING_8)
+ red_increase_wrong (s: STRING_8)
+ red_increase_exception (s: STRING_8)
+ yellow_increase_exception (s: STRING_8)
+ make_gray
+ gray_increase_ignore (s: STRING_8)
+ ignore_table (s: STRING_8)
+ set_more (p: ES_HTML_PARSER)
```

Figure 5.4: Features available in `ES_HTML_PARSE`

5.1.2 Dealing with the Fixture code

As we discussed in earlier chapters, The Fit framework provides three standard types of Fixtures: *Column Fixture*, *Action Fixture*, and *Row Fixture*. In addition, Fit allows the developers to implement new types of Fixtures.

In ES-Fit, various types of Fixtures (i.e., `ES_COLUMN_FIXTURE`, `ES_ACTION_FIXTURE`, and `ES_ROW_FIXTURE`) are implemented by inheriting from `ES_FIXTURE_UNIT` class (see BON diagram in Fig. 5.5). This class contains all the core features to implement and run various types of Fixtures. Each `ES_FIXTURE_UNIT` object correspond to a table in the HTML input file. `ES_FIXTURE_UNIT` contains a mapping which binds the string names of the operations defined in the HTML table, to the `ES_FIXTURE_CASE` objects. This mapping can be accessed through the `bindings` feature of `ES_FIXTURE_UNIT`.

`ES_FIXTURE_CASE` objects represent a *basic unit of computation* that can be performed by ES-Fit and refer to a cell in the HTML table that has a current (original) value and the future (computed) value. Each `ES_FIXTURE_CASE` object could be attached to a cell of the current HTML table.

This is done through the `set_fixture_table_cell` feature. Therefore, if the computation is a function (returns a value) the result of that can be directly reflected back the corresponding cell (e.g., by changing the color of the cell).

As shown in Fig. 5.5, there are two kinds of Fixture cases: one that returns a value, i.e., ES_FIXTURE_CASE_FUNCTION, and one that does not return a value (ES_FIXTURE_CASE_PROCEDURE). The original string value of a particular cell can be accessed using the fixture_case_table_value feature. The mapping between an ES_FIXTURE_CASE object and table name is done manually by the developer in the Fixture code using the bind feature.

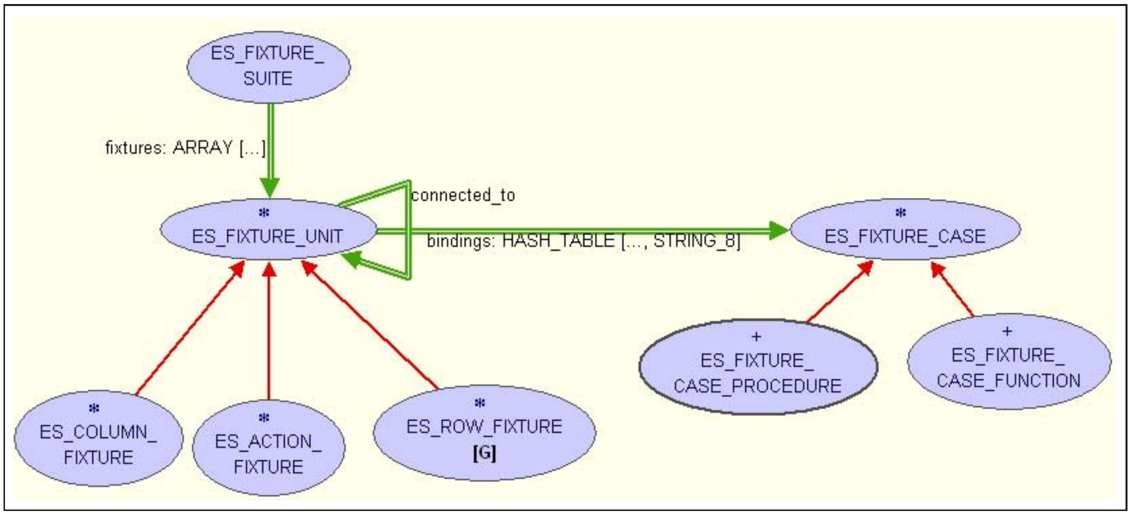


Figure 5.5: ES-Fit Fixture architecture

5.1.3 Flexible Fixture redefinition

In order to have an object-oriented and flexible way to define Fixtures, we decided to make the ES_FIXTURE_UNIT a deferred class (see Fig. 5.5). This allows the developers to implement various Fixture types by simply inheriting from

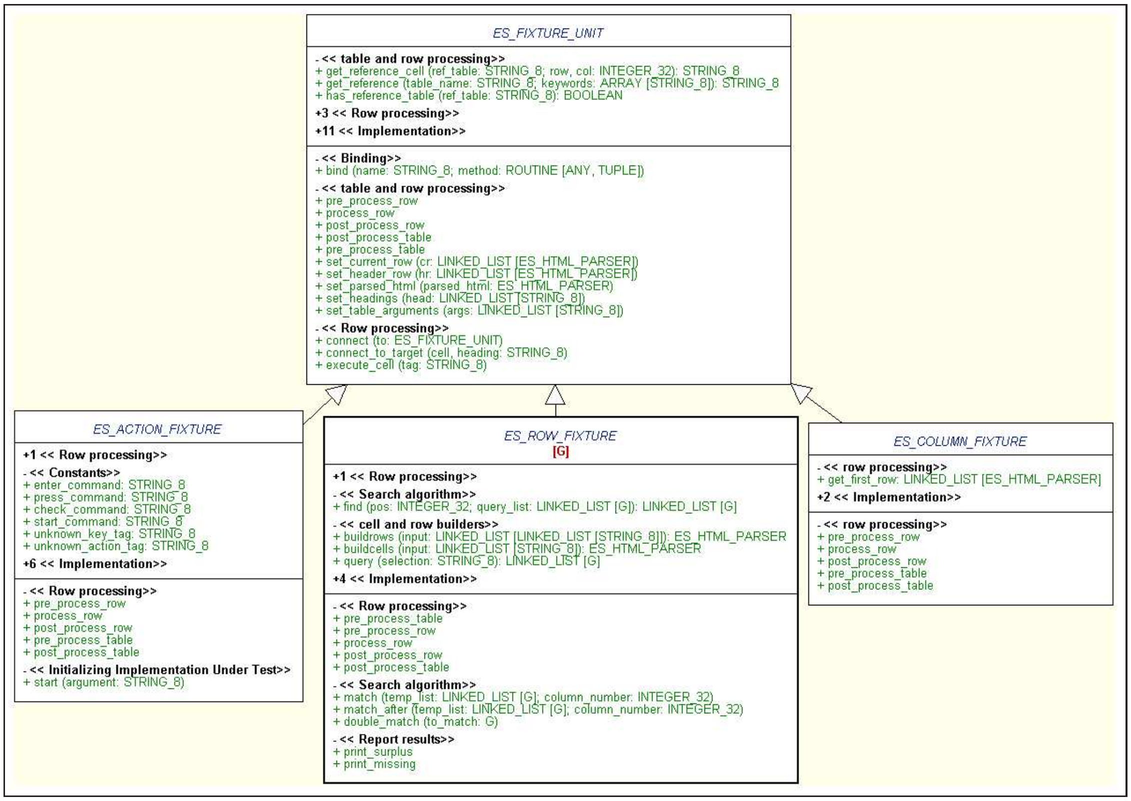

ES_FIXTURE_UNIT
- << table and row processing>>
+ get_reference_cell (ref_table: STRING_8; row, col: INTEGER_32): STRING_8
+ get_reference (table_name: STRING_8; keywords: ARRAY [STRING_8]): STRING_8
+ has_reference_table (ref_table: STRING_8): BOOLEAN
+3 << Row processing>>
+11 << Implementation>>
- << Binding>>
+ bind (name: STRING_8; method: ROUTINE [ANY, TUPLE])
- << table and row processing>>
+ pre_process_row
+ process_row
+ post_process_row
+ post_process_table
+ pre_process_table
+ set_current_row (cr: LINKED_LIST [ES_HTML_PARSER])
+ set_header_row (hr: LINKED_LIST [ES_HTML_PARSER])
+ set_parsed_html (parsed_html: ES_HTML_PARSER)
+ set_headings (head: LINKED_LIST [STRING_8])
+ set_table_arguments (args: LINKED_LIST [STRING_8])
- << Row processing>>
+ connect (to: ES_FIXTURE_UNIT)
+ connect_to_target (cell, heading: STRING_8)
+ execute_cell (tag: STRING_8)
ES_ACTION_FIXTURE
+1 << Row processing>>
- << Constants>>
+ enter_command: STRING_8
+ press_command: STRING_8
+ check_command: STRING_8
+ start_command: STRING_8
+ unknown_key_tag: STRING_8
+ unknown_action_tag: STRING_8
+6 << Implementation>>
- << Row processing>>
+ pre_process_row
+ process_row
+ post_process_row
+ pre_process_table
+ post_process_table
- << Initializing Implementation Under Test>>
+ start (argument: STRING_8)
ES_ROW_FIXTURE
[G]
+1 << Row processing>>
- << Search algorithm>>
+ find (pos: INTEGER_32; query_list: LINKED_LIST [G]): LINKED_LIST [G]
- << cell and row builders>>
+ buildrows (input: LINKED_LIST [LINKED_LIST [STRING_8]]): ES_HTML_PARSER
+ buildcells (input: LINKED_LIST [STRING_8]): ES_HTML_PARSER
+ query (selection: STRING_8): LINKED_LIST [G]
+4 << Implementation>>
- << Row processing>>
+ pre_process_table
+ pre_process_row
+ process_row
+ post_process_row
+ post_process_table
- << Search algorithm>>
+ match (temp_list: LINKED_LIST [G]; column_number: INTEGER_32)
+ match_after (temp_list: LINKED_LIST [G]; column_number: INTEGER_32)
+ double_match (to_match: G)
- << Report results>>
+ print_surplus
+ print_missing
ES_COLUMN_FIXTURE
- << row processing>>
+ get_first_row: LINKED_LIST [ES_HTML_PARSER]
+2 << Implementation>>
- << row processing>>
+ pre_process_row
+ process_row
+ post_process_row
+ pre_process_table
+ post_process_table

Figure 5.6: Available features of ES_FIXTURE_UNIT

the ES_FIXTURE_UNIT class and effecting its deferred features. This architecture abstracts away unnecessary loop structures from the Fixture code. The benefit is that the developers do not have to worry about the implementation details of running a Fixture table. This is because the underlying ES-Fit engine reads and processes *one row at a time*. Fit programmers need only to figure out how they would like ES-Fit to process one row at a time instead of thinking about the whole table.

The developer may decide to inherit from any of the standard fixtures or start writing the code from scratch. If we inherit from a standard Fixture, we can change the default behavior of that Fixture by redefining the following key methods related to table processing. A list of the available features is shown in Fig. 5.6. The complete source code is available at the ESpec website. Each Fit table is processed as follows:

- **pre_process_table:** actions to be performed before executing the current *table*.
- **pre_process_row:** actions to be performed before executing the current *row* of the current *table*.
- **process_row:** actions to be performed while executing the current *row* of the current *table*.
- **post_process_row:** actions to be performed after the execution of the current *row* of the current *table*.
- **post_process_table:** actions to be performed after the execution of the current *table*.

Fig. 5.7 illustrates how `ES_FIXTURE_UNIT` processes the tables. In the following sections we describe how various kinds of default Fixtures are implemented using this architecture.

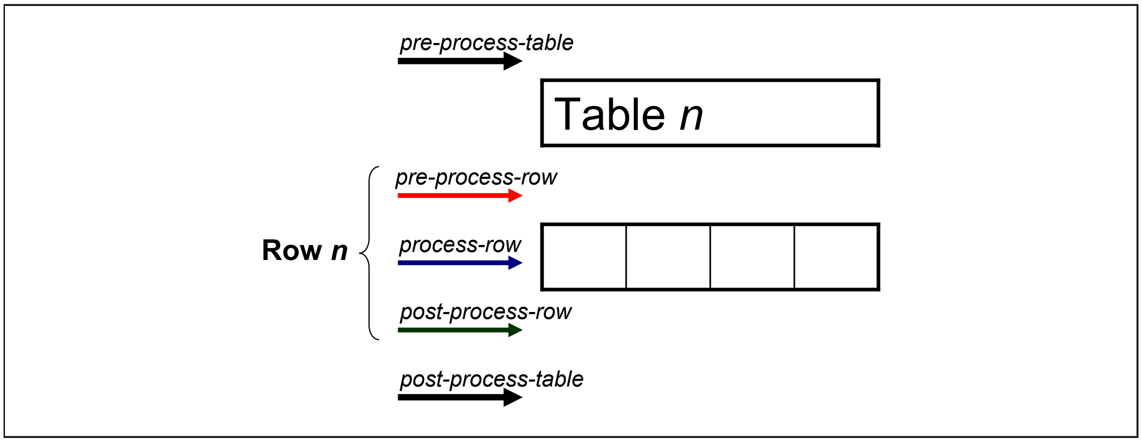


Figure 5.7: How ES-Fit processes the tables

5.1.4 Implementing standard Fixtures

Column Fixture: The Column Fixture is processed row by row. The concept is that each column corresponds to a separate call to the Fixture, either to store a table value for later use, or to provide a calculated result using the previously stored values.

The class `ES_COLUMN_FIXTURE`, provides facilities for the developers to define a Column Fixture. In order to develop `ES_COLUMN_FIXTURE`, we inherit from `ES_FIXTURE_UNIT` and redefine the above key features to generate the column Fixture behavior (see Fig. 5.6). For example, before processing a row of the table, we must process the header information of the table to generate `ES_FIXTURE_CASE` objects. This is done by redefining the `pre_process_row` feature. The Fixture code for the `ES_COLUMN_FIXTURE` is located at the ESpec website.

Action Fixture: The Action Fixture is used to run a sequence of actions on the underlying application. As in all Fixtures, the first row contains one cell that specifies the name of the Fixture. The rest of the table consists of rows which each have three columns. The first column contains one of four operations and the second and third columns contain actions and data for the operation.

For most of the operations, the second column contains the name of a field, method or property, and the third field contains data to be set or checked.

In order to develop `ES_ACTION_FIXTURE`, we inherit from `ES_FIXTURE_UNIT` and redefine the above key features to generate the action Fixture behavior (see Fig. 5.6). The Fixture code for the `ES_ACTION_FIXTURE` is located at the ESpec website.

Row Fixture: A Row Fixture associated with a table tests whether the expected elements of a list (or database) matches the actual elements in the list (or database). The developer creates a Row Fixture by inheriting from class `ES_ROW_FIXTURE[G]`, where `G` is a generic parameter which must be instantiated to the type of the object in the database. As before, the table headings are bound (via agent expressions) to appropriate routines. A deferred function routine query must be effected by the developer. The query routine returns a linked list representing the items in the database of the business logic. An algorithm matches rows with objects based on one or more keys. Objects may be missing or in surplus and are so noted [68]. The Fixture processes all the rows of one table following steps:

1. **Bind:** which is done manually by binding the table headings to agents defined in the Fixture code in the `make` routine.
2. **Query:** gets the list of *objects* from the System Under Test.
3. **Match:** compares the expected elements and the list of objects returned by the `query` feature.
4. **Build:** creates html for missing rows.
5. **Mark:** marks missing and surplus rows as such.

The comparison is made using a recursive partitioning strategy. That is, on the first pass both the rows from the table and the objects from the collection are partitioned based on the first column. The values from the table are converted to object format before the partitioning (we will see more on the conversion in later section).

If a partition contains exactly one row and one object, all fields are compared and the results displayed and tabulated. Fields which do not compare are marked wrong, fields which do compare are not marked. If either side is empty, all members of the other are marked as containing either extra or missing rows.

If either side contains more than one row or object (and the other is not empty) then that partition is recursively partitioned on the next column. The process continues until either a match, extra or missing row is found, or we run out of columns. In the latter case we obviously have duplicates, so the top entries are matched, and the excess on either side is either extra or missing.

The match process ends when both sides contain exactly one object (a match), one side or the other contains no objects (`missing` or `surplus`) or there are no more columns to partition. This last case is only possible if there are duplicate rows or objects; as many rows and objects as possible are matched, the remainder are either missing or surplus. The results appear in the same order they occurred in the table. Extra rows are inserted immediately after the cluster they belong to.

Columns which are missing data are ignored in the match. Columns which do not match in any object are an error and the table is aborted immediately. Objects which have missing attributes during the match are otherwise marked as excess; missing attributes after the partitioning process completes are considered mismatches and are marked wrong.

In order to develop `ES_ROW_FIXTURE`, we need to inherit from `ES_FIXTURE_UNIT` and redefine the above key features to generate the action Fixture behavior. The Fixture code for the `ES_ROW_FIXTURE` is located at the ESpec website.

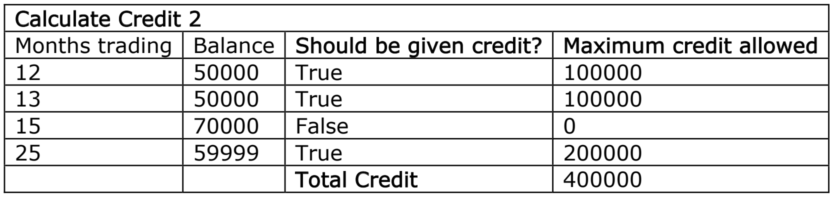
5.1.5 Implementing new Fixture types

Consider Table 5.2 which is an extension of our credit example from Chapter 3 (see Section E.2 for the Fixture code). The table looks like a Column Fixture, but it also has a *Total Credit* in the last row (as in a spreadsheet). Since the table is very similar to a Column Fixture, we inherit from ES_COLUMN_FIXTURE and redefine the standard behavior. We have redefined routines process_row, and post_process_table (lines 4–5). Routine process_row (lines 42–49) is redefined to ignore the last row as this row will be treated in feature post_process_table at which point it executes a routine that is associated with the rightmost bottom cell where the total of all credits must be calculated and checked.

In post_process_table (lines 51–55), we must (a) specify the target cell where the total credit will be printed, and (b) bind a routine to the cell (in this case routine credit_sum). We may think of the table as a spreadsheet with the appropriate cell at the intersection of row *Total Credit* and column *Maximum credit allowed* (Table 5.2). The specification of this cell is done by a call to connect_to_target at line 53. This routine inherited from ES_FIXTURE_UNIT. As before, the binding of the cell to routine credit_sum is done in the constructor make (lines 12–17). The credit_sum attribute is added to the Fixture class to calculate the required totals. Since we need the values in the last column of the table (with heading

"Maximum credit allowed"), we increase the credit_sum attribute in the agent that handles that column (i.e., credit_limit).

The row-by-row processing, adopted in this design, allows us to create new fixtures with relative ease.



Calculate Credit 2			
Months trading	Balance	Should be given credit?	Maximum credit allowed
12	50000	True	100000
13	50000	True	100000
15	70000	False	0
25	59999	True	200000
		Total Credit	400000

Table 5.2: New kind of Fit table

5.1.6 Execution of Fixtures

ES-Fit can execute on either an input HTML file or an input directory containing many HTML files. In directory mode, the system will recursively run the parser on every file in the directory that has an "HTML" or "HTM" extension. After the HTML files are read and parsed, and the ES_FIXTURE_UNIT objects are matched with the corresponding tables in the document, then the Fixtures are executed.

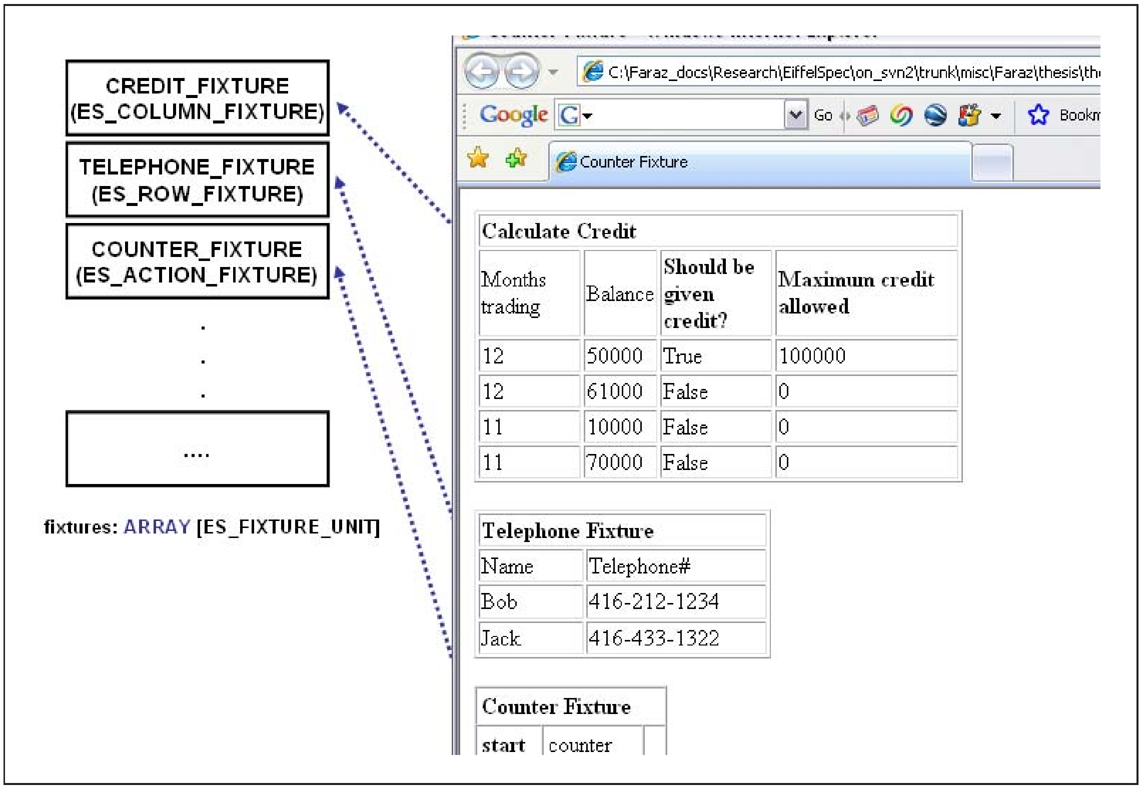
A single HTML document may contain as many Fit tables (with various types) as is needed. We associate an HTML file with an ES_FIXTURE_SUITE object. An instance of ES_FIXTURE_SUITE contains a collection of ES_FIXTURE_UNIT objects that can be accessed through the `fixtures` attribute[7]. The order in which `ES_FIXTURE_UNIT` objects are stored in the `fixtures` array correspond to the order in which Fit tables appear in the HTML document. These `ES_FIXTURE_UNIT` objects are added to the system by the developer in the creation routine in the root class of the system via routine `add_fixture`. This routine binds the name on the header of each HTML fit table to the corresponding objects in the suite. Fig. 5.8 shows how `ES_FIXTURE_UNIT` objects (stored in the `ES_FIXTURE_SUITE`) object correspond to the Fit tables in the HTML document. Fig. 5.8 shows that we can store various concrete subclasses of `ES_FIXTURE_UNIT` class (e.g., `CREDIT_FIXTURE`, `TELEPHONE_FIXTURE`, `COUNTER_FIXTURE`) in the `ES_FIXTURE_SUITE`.

As mentioned earlier, all Fixture objects associated with an HTML document is stored in `ES_FIXTURE_SUITE`. A routine called `run_html` in `ES_FIXTURE_SUITE` will go through the list of `fixtures` that are currently stored and execute them one after the other. For every table *T* in the HTML document (that has an associated `ES_FIXTURE_UNIT` object *O*), we execute the following steps on *O*:

1. call *pre-process-table*

2. while *T* has more rows

[7] Since `ES_FIXTURE_UNIT` is a deferred class, we cannot directly create an object of its type. Therefore, we need to create an object of a concrete subclass of `ES_FIXTURE_UNIT`, e.g. `ES_COLUMN_FIXTURE`, `ES_ROW_FIXTURE`, `ES_ACTION_FIXTURE`, etc...)



Calculate Credit			
Months trading	Balance	Should be given credit?	Maximum credit allowed
12	50000	True	100000
12	61000	False	0
11	10000	False	0
11	70000	False	0

Telephone Fixture	
Name	Telephone#
Bob	416-212-1234
Jack	416-433-1322

Counter Fixture		
start	counter	

Figure 5.8: `ES_FIXTURE_SUITE` and its relation to the HTML document

(a) call *pre-process-row*

(b) call *process-row*

(c) call *post-process-row*

3. call *post-process-table*

In `process-row` we execute the `ES_FIXTURE_CASE` objects which correspond to the basic calculations of a row. In order to execute an operation in `ES_FIXTURE_CASE`, we need to read the input values from the Fit tables and convert them to appropriate types (e.g., `BOOLEAN`, `STRING`) as required by the operation. The question that comes to mind is how to do this conversion automatically. The following section describes how this is done in ES-Fit.

5.1.7 Reflection

The original Fit framework in Java is highly dependent on the reflection capabilities of the supporting language[8]. The Eiffel language itself does not yet support reflection although external reflection libraries are available [61]. Reflection removes some of the burden on the developer. For example, in a Column Fixture, a class is automatically created via reflection with the same name as the table heading. In that class, attributes and methods corresponding to the column headings are created automatically. All that the developer needs to do is to implement the method referring to the attributes for what are, in effect, the arguments of the methods.

One of our design decisions was not to use the reflection capabilities. One reason was pragmatic—we were not sure at the time how robust the external libraries were. In addition, a major problem with reflection is that it forces the customer to use names consistent with Java syntax for class, attribute and method

[8] Reflection is a mechanism that allows an application to query its own metadata. Reflection allows an application to discover information about itself so that it may display this information to the user, modify its own behavior by using late-binding and dynamic invocation (i.e., binding to and calling methods at runtime), or create new types at runtime (Reflection Emit).

names. In our approach, the developer has to do more work (by binding names to routines) but this has the advantage that customers have complete freedom in the choice of names consistent with their business logics. We will describe other advantages to our approach.

Eiffel does provide some reflection capabilities. Agent expressions encapsulate routines as objects so that those routines can be executed at some later time as needed. Class `INTERNAL` in the base library offers several features to access and manipulate the state of an object and to create new instances of a particular type. The Fit framework benefits from reflection in the following scenarios:

1. When creating a new instance of the Fixture class based on the class name provided in the HTML file (table header).

2. When binding the string name of the method or attribute in HTML file to the actual Methods or Fields in the Fixture class.

3. When setting Fields or executing Methods of the Fixture class on the converted input values read from the table.

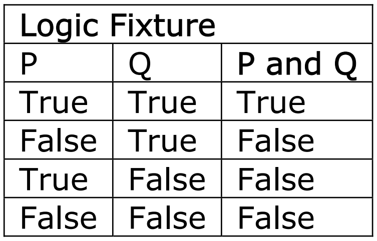
4. When comparing the result of method calls with the contents of a cell in the HTML file.

As mentioned in the above sections, in ES-Fit we take care of (1) by using the agent mechanism of Eiffel where developers create the Fixture objects in the root class and bind those objects to the names that appear in the HTML file (using the add_fixture routine). To handle (2) we use agents to encapsulate various operations of each Fixture. ES_FIXTURE_CASE class is used to pass these operations to ES-Fit engine. Binding the actual names of these operations to these ES_FIXTURE_CASE objects is done by the developers in the make routine of the Fixture class (using the bind routine).

This restriction comes with a valuable benefit: we are able to use the customer provided HTML tables directly without further modifications. In other words, ES-Fit is neither dependent upon the strings in the header of the tables nor on the string names of the operations. This is valuable because the customer can focus on the business logic without having to think about underlying classes and methods.

The original Java version of Fit stored column inputs in attributes and column computations in methods without arguments. In our approach, the routines have arguments corresponding to the input columns so that the routine better encapsulates the computations.

In order to deal with scenarios (3) and (4), we need to find a way to convert the input string values (that are read from the HTML file) to the actual types of the corresponding agent arguments. Note that this conversion should be done at run-time. For example, consider the Fit table 5.3 and its Fixture code in Fig. 5.9.



Logic Fixture		
P	Q	P and Q
True	True	True
False	True	False
True	False	False
False	False	False

Table 5.3: A Column Fixture

Before executing the `calculate_and` agent, ES-Fit has to convert the input values (read from the tables cells) to the appropriate type conforming to the type of the argument (`BOOLEAN` in this case) and use these values as input arguments. Similarly, after the operation is finished, ES-Fit has to convert the expected output value of the agent (read from the table) to the return type of the `calculate_and` agent (`BOOLEAN` in this case).

5.1.8 Type Conversion

The type conversions of ES-Fit use the Eiffel tuple and agent (see Appendix A) constructs.

The tuple type is any type based on class `TUPLE`, i.e., any type of the form `TUPLE` $[T_1, T_2, ..., T_n]$ for any n (including 0, for which there is no generic parameter). An instance of `TUPLE` $[T_1, T_2, ..., T_n]$ is a tuple whose first element is an instance of T_1, the second element being an instance of T_2 etc. Mathematically, `TUPLE` $[T_1, T_2, ..., T_n]$ is the set $TUPLE_n$ of all partial functions f from N+

```
class LOGIC_FIXTURE inherit
    ES_COLUMN_FIXTURE

create
    make

feature{NONE}

    make
        do
            bind ("P and Q", agent calculate_and)
        end

    calculate_and (a, b: BOOLEAN): BOOLEAN
        do
            Result := a and b
        end
end -- class LOGIC_FIXTURE
```

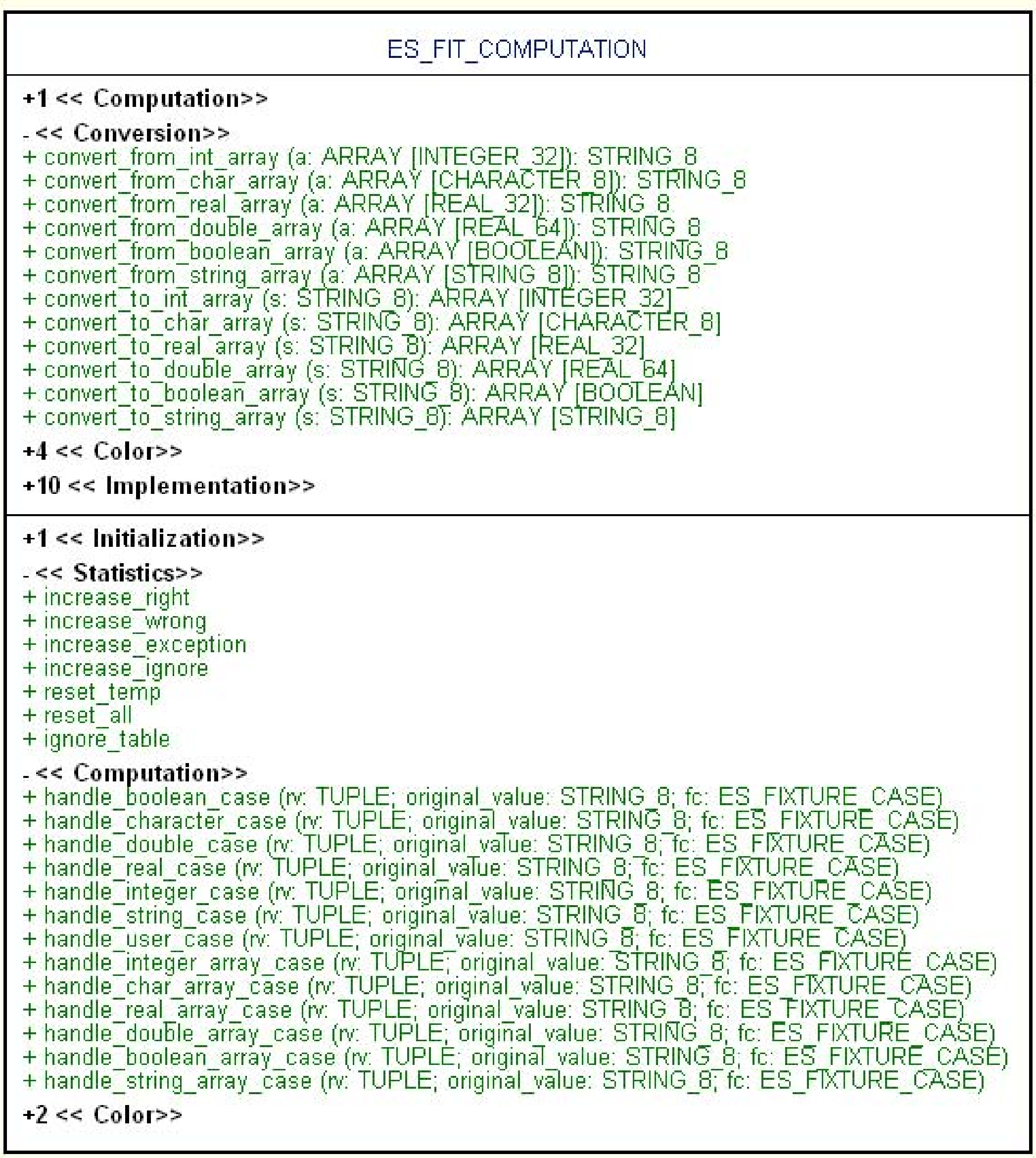
Figure 5.9: The Fixture code associated with a Column Fixture 5.3

(the set of non-negative integers) to $T_1 \cup T_2 \cup ...T_n$, such that: (a) The domain of f contains the interval $1 \cdots n$ (in other words, f is defined for any i such that $1 \leqslant i \leqslant n$). (b) For $1 \leqslant i \leqslant n$, $f(i)$ is a member of T_i. There can be more than n elements to the tuple. For example, the tuple $[5, ``foo'']$ (i.e., the tuple with first element 5 and second element “foo”) is an instance of all of the following tuple types: `TUPLE`; `TUPLE [INTEGER]`; `TUPLE [INTEGER, STRING]`.

Developers link Eiffel routines (with arguments) to computations that must be performed in Fit tables. The binding operation stores the routines as an agent expression object (an instance of `ROUTINE`) which can be queried for its empty operands as a tuple (via the query `empty_operands`). The tuple so returned (representing the arguments of the routine) can be queried for the type of each operand. For example, class TUPLE has an is_integer_item(i) query that may be used to check if the *i*-th argument is of type integer. This type checking is done in class ES_FIXTURE_CASE.

The type of the output of the computation is detected in the similar way (output type is converted to a TUPLE object and then checked against the various supported types).

After detecting the input argument/output types, we need to convert the corresponding string value that was read from the HTML table, to the detected type (for example, if the first argument was of type INTEGER, the value read from the table should be converted to an INTEGER). If the there is an error in conversion, it will be reported to the table as a *type error* (for example, if we expect to convert to an INTEGER, and the read value was a character). Type conversion is done in the ES_FIT_COMPUTATION class which has features for converting STRING type to various supported types (see Fig. 5.10). These features are of form handle_T_case (where T is the name of the type, e.g. integer). ES-Fit currently supports the following types: STRING, INTEGER, REAL, BOOLEAN, CHARACTER, ARRAY [STRING], ARRAY [INTEGER], ARRAY [REAL], ARRAY [BOOLEAN], and ARRAY [CHARACTER].



ES_FIT_COMPUTATION

+1 << Computation>>

- << Conversion>>
+ convert_from_int_array (a: ARRAY [INTEGER_32]): STRING_8
+ convert_from_char_array (a: ARRAY [CHARACTER_8]): STRING_8
+ convert_from_real_array (a: ARRAY [REAL_32]): STRING_8
+ convert_from_double_array (a: ARRAY [REAL_64]): STRING_8
+ convert_from_boolean_array (a: ARRAY [BOOLEAN]): STRING_8
+ convert_from_string_array (a: ARRAY [STRING_8]): STRING_8
+ convert_to_int_array (s: STRING_8): ARRAY [INTEGER_32]
+ convert_to_char_array (s: STRING_8): ARRAY [CHARACTER_8]
+ convert_to_real_array (s: STRING_8): ARRAY [REAL_32]
+ convert_to_double_array (s: STRING_8): ARRAY [REAL_64]
+ convert_to_boolean_array (s: STRING_8): ARRAY [BOOLEAN]
+ convert_to_string_array (s: STRING_8): ARRAY [STRING_8]

+4 << Color>>

+10 << Implementation>>

+1 << Initialization>>

- << Statistics>>
+ increase_right
+ increase_wrong
+ increase_exception
+ increase_ignore
+ reset_temp
+ reset_all
+ ignore_table

- << Computation>>
+ handle_boolean_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_character_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_double_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_real_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_integer_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_string_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_user_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_integer_array_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_char_array_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_real_array_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_double_array_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_boolean_array_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)
+ handle_string_array_case (rv: TUPLE; original_value: STRING_8; fc: ES_FIXTURE_CASE)

+2 << Color>>

Figure 5.10: Available features of ES_FIT_COMPUTATION

5.1.9 Dealing with Contract Violations

If there are violations of any kind (e.g., Contract violations or assertion violations, etc.) ES-Fit reports it directly to the table (with yellow background) with a detailed description indicating the location of the error. This information can be used by the developers as a debugging facility to locate bugs in the specification and/or the implementation.

In order to be able to report such errors, we need to use the Base Library class `EXCEPTIONS`, which provides facilities such as reporting the type of the last exception. The `EXCEPTIONS` class provides the symbolic names for all exceptions (e.g., "precondition violation", "post condition violation", etc.). We can then detect various exceptions by testing the generated exception against various possibilities. In ES-Fit, the `ES_FIXTURE_CASE` class inherits from the `EXCEPTIONS` class and deals with run-time violations.

5.1.10 Result comparison

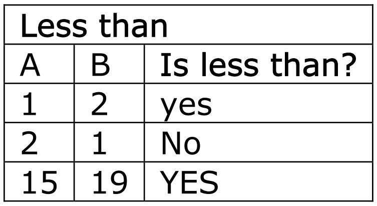
In this step, we compare the value that was returned by the business logic (actual) to the value which was read from the table (expected). The comparison is done in the `ES_FIT_COMPUTATION` class right after the conversion. If the expected value is different than the actual value, a report is written back to the corresponding table cell (with red background), otherwise, the report shows a "passed" case (with green background). These statistics are also kept in the `ES_FIT_COMPUTATION` class.

If the output cell from the table is left as blank then the cell will be ignored and the actual result will be written to that cell (with gray background).

The **Void** keyword is used in the Fit tables to expect a void result. In this case, the cell will be marked as green if the returned value by the system under test is void. The **Empty** keyword can be used in the Fit table to express expectation for an empty string (i.e., ""). In this case, the cell is marked green only if the resulting string generated by the system under test is empty. The **Error** keyword is used when an exception is expected (e.g, a precondition violation). The corresponding cell will be marked as green only if during the execution of the agent a violation happen.

ES-Fit also allows the developer to define new types and new equality rules. The new equality rule will be used by the ES-Fit engine to check the expected value (read from the table) against the actual value (returned by the system). For example, the developer may associate string "Yes" from the Fit table to Boolean value "True".

`ES_FIT_TYPE` class can be used to define new type definitions. The developer has to create an object of type `ES_FIT_TYPE` and define a new equality rule (e.g.,



Less than		
A	B	Is less than?
1	2	yes
2	1	No
15	19	YES

Table 5.4: An example of table with developer defined types

how two values are equal) and pass it to ES-Fit. ES-Fit takes care of the rest.

A simple example is shown in Table 5.4. The first two columns are the input integer values and the third column is the expected output value. As shown, the customer has used "Yes" instead of "True" and "No" instead of "False" values.

Fig. 5.11 shows the Fixture code that deals with Table 5.4. The agent that is bound to computation $a < b$, returns an object of type `ES_FIT_TYPE` (line 12); this object is created at line 15. The new equality rule which will be used by ES-Fit to compare the actual and the expected values is defined at lines 24–27 in a form of an agent function which takes two string arguments and returns a Boolean value. In our example, the developer decided that the comparison should be done case insensitive (see line 26). The equality agent is then passed to the `ES_FIT_TYPE` object using the `set_comparison_func` feature. The developer can then manually convert the actual value (returned by the business logic) to the desired strings, e.g., when $a < b$, the developer tells ES-Fit to interpret this as a "Yes" (see line 18).

```
class LESS_THAN_FIXTURE inherit
    ES_COLUMN_FIXTURE
create make

feature {NONE}

    make
        do
            bind ("Is less than?", agent compute)
        end

    compute(a, b: INTEGER): ES_FIT_TYPE -- returns an ES_FIT_TYPE
            -- is a less than b?
        do
            create Result
            Result.set_comparison_func (agent equality)
            if a < b then -- binding "Yes" to true
                Result.set_actual_value ("Yes")
            else -- binding "No" to false
                Result.set_actual_value ("No")
            end
        end

    equality (s1, s2: STRING): BOOLEAN is
            -- Defines how the comparison is done by ES-Fit
        do
            Result := s1.is_case_insensitive_equal (s2)
        end
end -- class LESS_THAN_FIXTURE
```

Figure 5.11: The Fixture code associated with Table 5.4

The type definition is useful because a customer can freely assert business needs without having to know much about programming. Another good application of type definition is when we are comparing real numbers with many decimal places. The developer can define an ϵ value (e.g., $a < b$ iff $a - b < \epsilon$) for comparing the actual and expected values read from the table.

5.1.11 Reference tables in the HTML input

A Fit requirements document may contain many tables that need to share data. For example, a description of a bank dealing with foreign currencies may need to refer to a table of standard conversion rates. In such a case it is convenient to place all the conversion rates in a single table with the data in that table shared by all the other tables. To update the conversions we need only make the change in one place. The standard Fit framework does not supply such a construct. We thus introduce such tables representing shared data with a new keyword **Reference**.

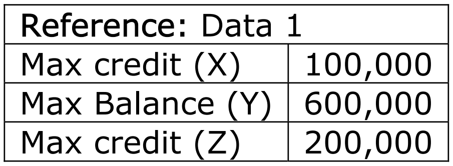
Consider the following requirement for calculating the credit limit of a company:

> **[R2]** Credit is allowed, up to an amount of $X for companies who have been trading with us for at least one year and have a balance owing of less than $Y. This credit is extended to an amount of $Z for companies who have been with us for more than two years.

The values X, Y and Z could be described via extra columns in Table 3.1; however, the intention in this case is that X, Y and Z are global data. It would be inconvenient to change these parameters for every row in the table. What we need is another table that contains this global data that is referenced by Table 3.1.

Customers may easily change the Reference Table to test that the code is working correctly with the new requirements. The standard Fit framework does not accommodate such references. We have thus extended the ES-Fit with the **Reference** keyword (see Table 5.5). Fixture code may refer to the Reference Table for retrieving the global data. The following code is used to refer to the *Max Balance (Y)* constant in the Reference Table “Data 1” (see line 37–40 in Fig. 5.12):

```
get_reference (‘‘Data 1’’, <<‘‘Max Balance (Y)’’, ‘‘?’’>>).to_real
```



Reference: Data 1	
Max credit (X)	100,000
Max Balance (Y)	600,000
Max credit (Z)	200,000

Table 5.5: Reference table for the credit example

The above expression searches a reference table with heading *Data 1* for a row that starts with *Max Balance (Y)*. The question mark (“?”) represents the value in the associated table cell that we wish to retrieve (i.e., $600,000). Since the value is read in form of a string, the developer has to convert it to a real value (using `to_real` feature).

The `get_reference` feature is implemented in `ES_FIXTURE_UNIT` in ES-Fit library and is available to all Fixture classes (see Fig. 5.6). Another important feature which can be used refer to a Reference Table is `get_reference_cell(t, n, m)` which returns the contents of the table cell located at row `n` and column `m` of the Reference Table `t`.

```
class CREDIT_FIXTURE inherit
  ES_COLUMN_FIXTURE
create
  make

feature {NONE}
  make
    do
      bind ("Should be given credit?", agent allow_credit)
      bind ("Maximum credit allowed", agent credit_limit)
    end

  allow_credit (months: INTEGER; balance: REAL): BOOLEAN
    local rules: RULE -- RULE class is part of business logic
    do
      create rules.make (max_balance, max_credit_1, max_credit_2)
      Result := rules.is_allowed (months, balance)
    end

  credit_limit (months: INTEGER; balance: REAL): REAL
    local rules: RULE
    do
      create rules.make (max_balance, max_credit_1, max_credit_2)
      Result := rules.credit_limit (months, balance)
    end

  max_credit_1: REAL is
    do
      Result := get_reference ("Data 1",<<"Max credit (X)", "?">>).to_real
    end

  max_credit_2: REAL is
    do
      Result := get_reference ("Data 1",<<"Max credit (Z)", "?">>).to_real
    end

  max_balance: REAL is
    do
      Result := get_reference ("Data 1",<<"Max Balance (Y)", "?">>).
          to_real
    end
end -- class CREDIT_FIXTURE
```

Figure 5.12: Referring to a Reference Table

5.1.12 Output

ES-Fit uses the `ES_HTML_PARSER` class to write the results back to the tables. There are a variety of features available to write output back to the tables. These features are implemented in the `ES_HTML_PARSER` class (see ESpec code for more information).

The following section describes the design and implementation of ES-Test.

5.2 ES-Test Architecture

ES-Test is the first lightweight unit testing framework for Eiffel. Unit testing, when combined with DbC, BON and other best practices, leads to rapid software development without sacrificing proper design principles. The BON static diagram of the `ES_TEST` cluster is shown in Fig. 5.13. This section introduces the basic architecture ES-Test.

The `ES_TEST` cluster contains three clusters:

- `CASES`
- `COLLECTIONS`
- `HTML-REPORTING`

ES-Test is flexible in that one can structure the tests in an arbitrary ordering.

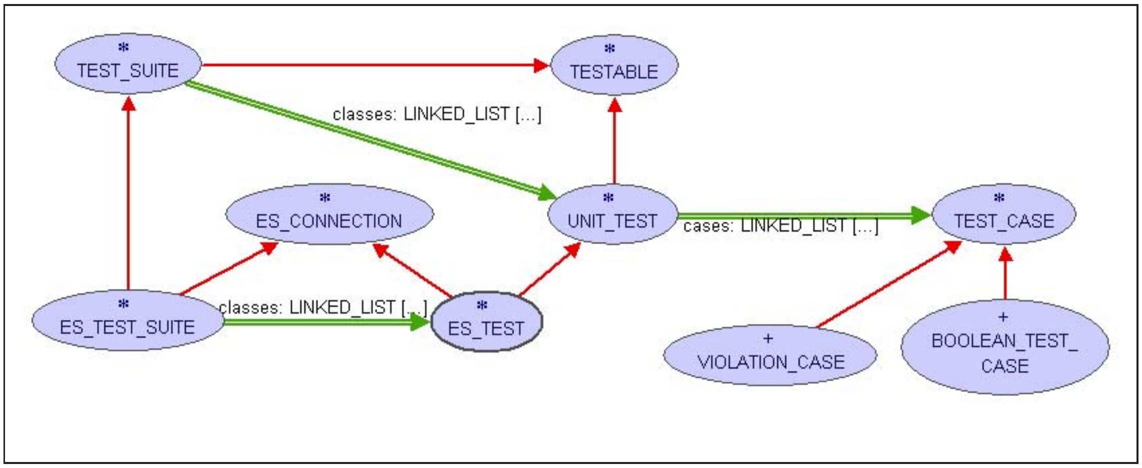


Figure 5.13: BON Static Diagram of es-test cluster located in ESpec library

To implement a unit test class, one is required to subclass UNIT_TEST and define zero or more test cases (see Chapter 2 for examples). The UNIT_TEST class contains a LINKED_LIST containing various test cases. The test cases are either a BOOLEAN_TEST_CASE or a VIOLATION_TEST_CASE (both inherit from TEST_CASE class). These test cases are added to the unit test objects directly by the developers in the concrete subclasses of UNIT_TEST. The test cases are added using any of the following methods: add_boolean_case, add_violation_case, and add_violation_case_with_tag. Zero or more UNIT_TEST subclasses are then aggregated into a subclass of TEST_SUITE which is then executed to run all tests.

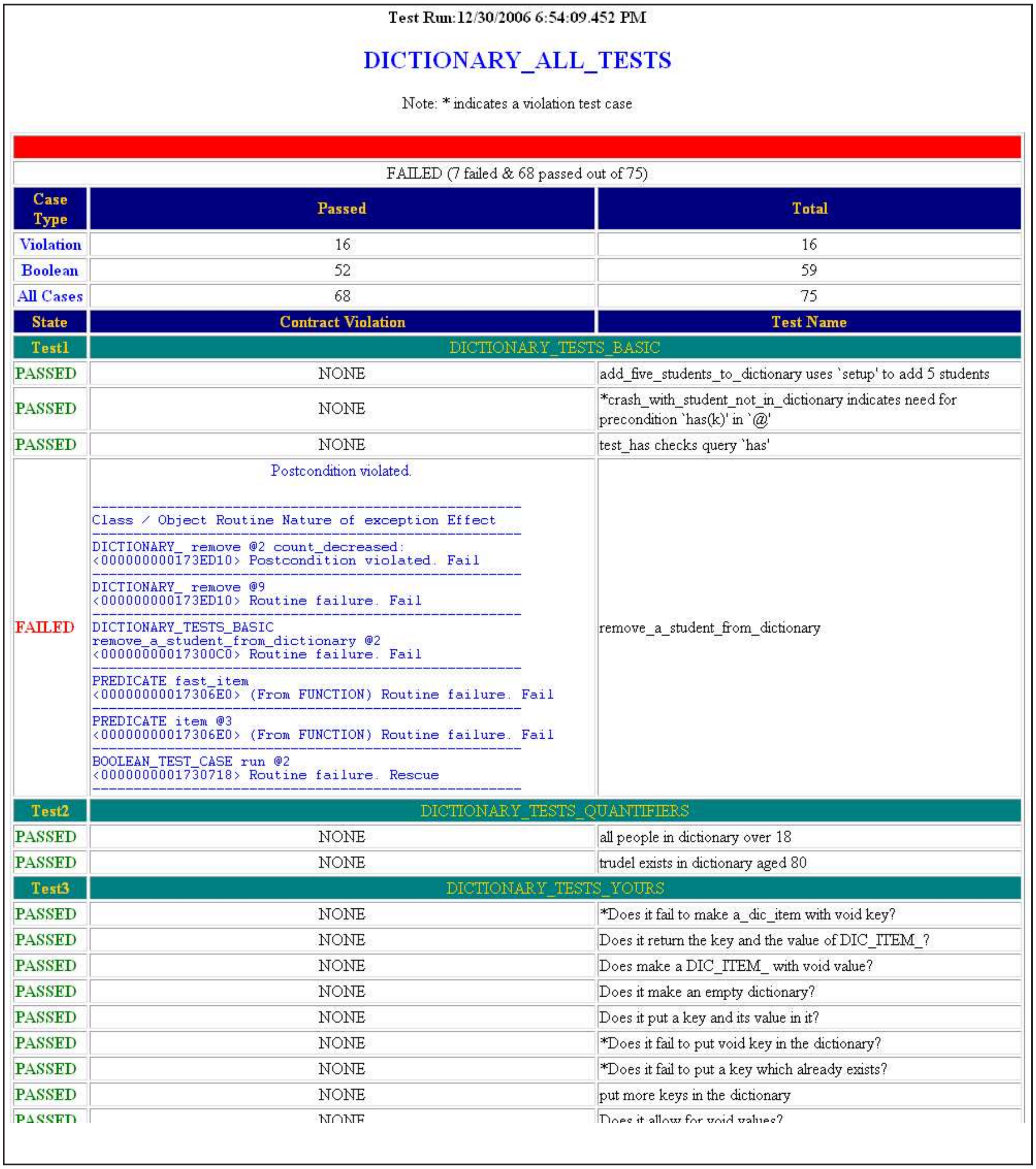
In summary, a TEST_SUITE object is a collection of UNIT_TEST objects which are themselves collections of TEST_CASE objects. The test suite is built up recursively: class tests and an inter-class test comprise a cluster test; cluster tests and an inter-cluster test comprise a system test. In all Unit Tests, the test cases will be added in the `make` creation routine. These cases were explained with examples in Chapter 2.

ES-Test supports two forms of output: HTML or GUI. The `HTML_REPORTING` cluster contains classes which implement HTML functionality. An HTML report will be generated for every unit test class showing the status of testing of each test case in that class. An example of an HTML output for a suite of Unit Tests (i.e., `ALL_DICTIONARY_TESTS`) is shown in Fig. 5.14. Violations are reported to this HTML report with information regarding the location of the violation. Contract violations can then be traced using the HTML output. The name of the HTML file is declared in the `make` creation feature of system root class.

5.2.1 Communication with ESpec's GUI

Running the tests (ES-Fit or ES-Test) can be done from either command-line (results in generation of an HTML output file), or from the ESpec GUI. The ESpec GUI helps developers to execute various modules of ESpec (i.e., ES-Fit, ES-Test or ES-Verify) individually or in unison under a single green/red bar.

The GUI communicates with the ESpec library through a TCP/IP stream (see Fig. 5.1). This design is chosen because it completely separates the business logic (ESpec library) from the GUI. The design also allows us to update the ESpec



Test Run:12/30/2006 6:54:09.452 PM

DICTIONARY_ALL_TESTS

Note: * indicates a violation test case

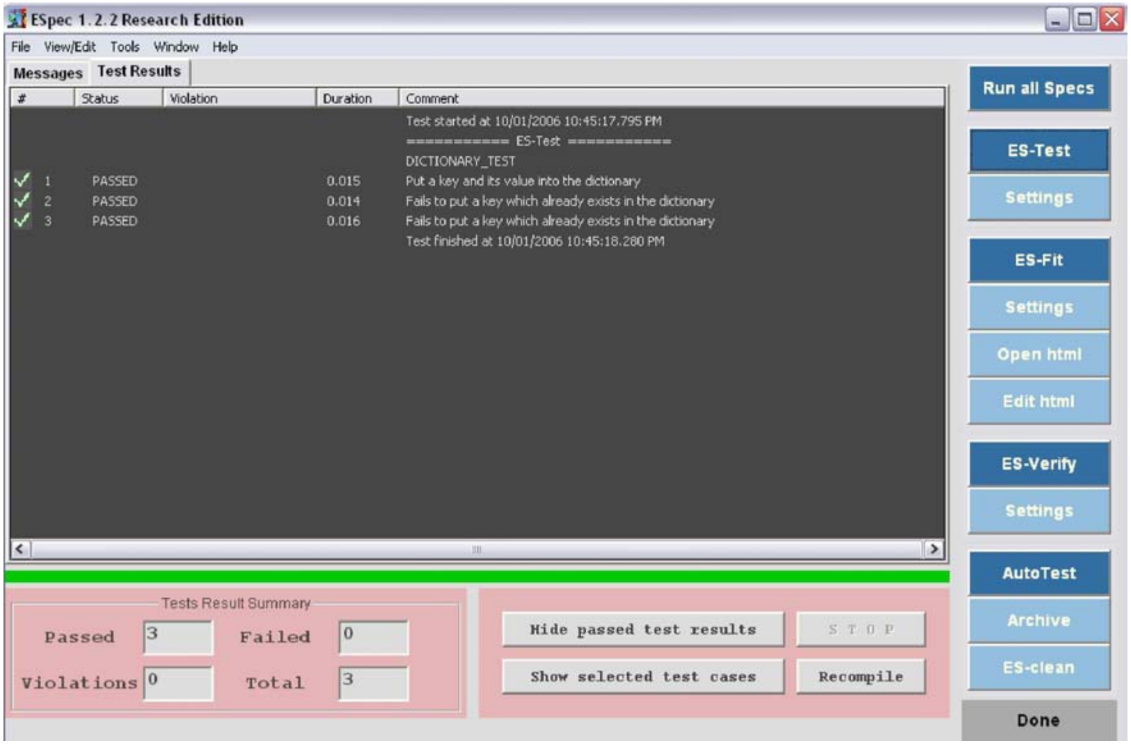
FAILED (7 failed & 68 passed out of 75)

Case Type	Passed	Total
Violation	16	16
Boolean	52	59
All Cases	68	75
State	**Contract Violation**	**Test Name**
Test1	DICTIONARY_TESTS_BASIC	
PASSED	NONE	add_five_students_to_dictionary uses `setup' to add 5 students
PASSED	NONE	*crash_with_student_not_in_dictionary indicates need for precondition `has(k)' in `@'
PASSED	NONE	test_has checks query `has'
FAILED	Postcondition violated. --- Class / Object Routine Nature of exception Effect --- DICTIONARY_ remove @2 count_decreased: <00000000173ED10> Postcondition violated. Fail --- DICTIONARY_ remove @9 <00000000173ED10> Routine failure. Fail --- DICTIONARY_TESTS_BASIC remove_a_student_from_dictionary @2 <00000000017300C0> Routine failure. Fail --- PREDICATE fast_item <00000000017306E0> (From FUNCTION) Routine failure. Fail --- PREDICATE item @3 <00000000017306E0> (From FUNCTION) Routine failure. Fail --- BOOLEAN_TEST_CASE run @2 <0000000001730718> Routine failure. Rescue ---	remove_a_student_from_dictionary
Test2	DICTIONARY_TESTS_QUANTIFIERS	
PASSED	NONE	all people in dictionary over 18
PASSED	NONE	trudel exists in dictionary aged 80
Test3	DICTIONARY_TESTS_YOURS	
PASSED	NONE	*Does it fail to make a_dic_item with void key?
PASSED	NONE	Does it return the key and the value of DIC_ITEM_?
PASSED	NONE	Does make a DIC_ITEM_ with void value?
PASSED	NONE	Does it make an empty dictionary?
PASSED	NONE	Does it put a key and its value in it?
PASSED	NONE	*Does it fail to put void key in the dictionary?
PASSED	NONE	*Does it fail to put a key which already exists?
PASSED	NONE	put more keys in the dictionary
PASSED	NONE	Does it allow for void values?

Figure 5.14: HTML report generated by ES-Test

library without having to re-compile the GUI.

The GUI runs the tests and displays the results. If all tests pass, a green bar is shown with test statistics (Fig. 5.15). Boolean/Violation test case successes are reported separately. As well, each test passed is listed. If even one test fails, a red bar is displayed and the failing tests are indicated (see Appendix D for GUI screenshots).



#	Status	Violation	Duration	Comment
				Test started at 10/01/2006 10:45:17.795 PM
				=========== ES-Test ===========
				DICTIONARY_TEST
✓ 1	PASSED		0.015	Put a key and its value into the dictionary
✓ 2	PASSED		0.014	Fails to put a key which already exists in the dictionary
✓ 3	PASSED		0.016	Fails to put a key which already exists in the dictionary
				Test finished at 10/01/2006 10:45:18.280 PM

Tests Result Summary

Passed 3 Failed 0

Violations 0 Total 3

Figure 5.15: ESpec runs ES-Test when "ES-Test" button is pressed

In order to communicate with the GUI, we have developed the `ES_CONNECTION` class. Various ESpec tools such as ES-Test, ES-Fit and ES-Verify use this class to send their results to the GUI. For example, `ES_TEST` class inherits from both `UNIT_TEST` (which is used to run Scenario Tests) and `ES_CONNECTION` allowing the ES-Test application to communicate with the ESpec GUI. Similarly, `ES_FIT` (which runs the Fit tests) and `ES_VERIFY` (which runs the verification tool) both inherit from `ES_CONNECTION` to report the results to the GUI.

`ES_CONNECTION` reports the test results immediately, as they are executed and controls the test progress with the ability to cancel a test run without losing the results. It opens a TCP client connection to the loopback interface (locahost) and sends the results to the GUI as soon as the information is available by the running application (because of this, ESpec library is dependent upon the `net` library in Eiffel Base). If a network error occurs while running the test suite, the execution stops before the next test case. This allows the GUI to abort test applications that are responsive (i.e., not in an infinite loop). Using this architecture, we can automatically change the behaviour of ESpec depending on the mode of execution (i.e., GUI or command line). `ES_CONNECTION` checks whether ESpec is running in GUI mode, if so, reports from various tools are sent to the GUI, otherwise, the report is sent to the command line. Fig. 5.16 shows a BON diagram of this part of ESpec library.

This architecture allows for future extension of ESpec (e.g., introducing model checking facility). The available features in `ES_CONNECTION` class are shown in

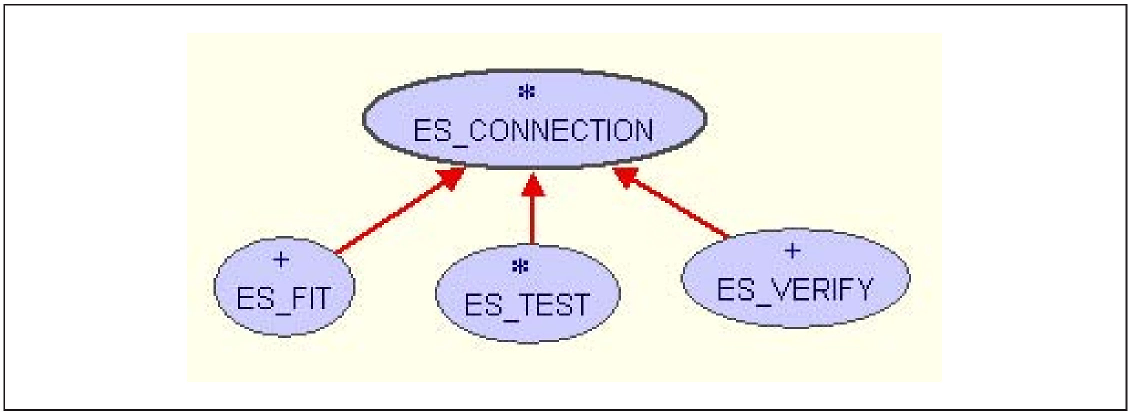


Figure 5.16: Connection architecture

Fig. 5.17.

5.3 ES-Verify

The design and implementation of ES-Verify is described in Appendix C. The remaining sections of this chapter are devoted to the description of ES_SUITE (the shared interface between various tools in the system).

5.4 Seamless integration of ES-Fit, ES-Test and ES-Verify

In Java, a developer may use JUnit [41] for Unit Tests and the Fit command line application for the Fit tests [31]. As pointed out earlier, ESpec adds to the standard tools some additional features. The first addition is that contracts are used to formally specify the details of the business logic. Violations of this specifica-

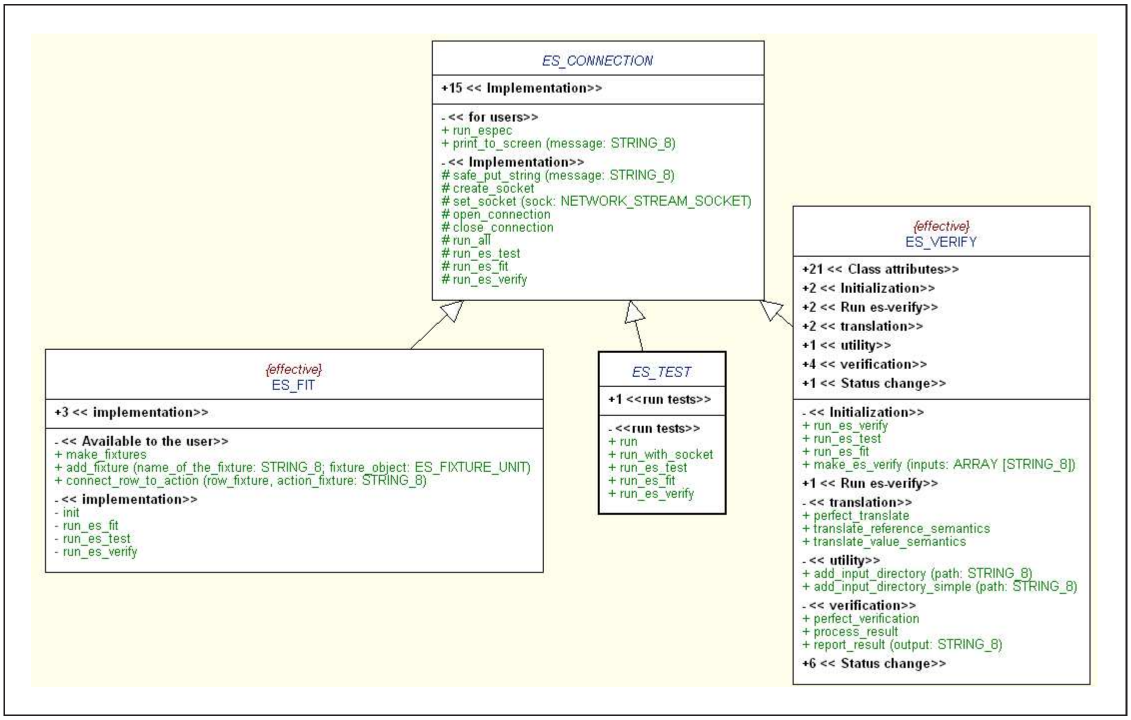


Figure 5.17: Available features of ES_CONNECTION

tion will be reported in the Scenario Tests and the Fit Tables. The second addition is that we unify the Fit Fixtures, Scenario Tests, and the Verification module in the same class, so that validation and verification can be performed simultaneously in order to certify the quality of the product.

In order to run the Fit tests, the software developer places the test Fixtures in a class that inherits from class ES_FIT. Scenario Tests are placed in a class that inherits from ES_TEST_SUITE. Similarly, to run ES-Verify, developer has to inherit from ES_VERIFY and specify the files to be verified. We may combine Validation (Fit Fixture Tests), Lightweight Verification (Scenario Tests and Contracts) and full verification (ES-Verify) under the report of a single green bar by declaring three types of tests in a class that is a descendant of ES_SUITE. Eiffel supports multiple inheritance, and thus allows us to develop this shared interface by inheriting from multiple classes in the system. We developed ES_SUITE which unifies the three types of tests in a single class by inheriting from ES_FIT, ES_TEST_SUITE and ES_VERIFY. So it would be enough that we inherit from ES_SUITE in system's root class where we combine various kinds of tests. Fig. 5.18 shows this architecture[9]

The list of available commands in the ES_SUITE class is shown in Table 5.6. These commands may be used in the make routine of a subclass of ES_SUITE.

[9] This architecture raises the *diamond problem*. In object-oriented programming languages with multiple inheritance, the diamond problem is an ambiguity that arises when two classes B and C inherit from a shared parent A, and another class D inherits from both B and C. It is called the "diamond" problem because of the shape of the class inheritance diagram in this situation. Class A is at the top, both B and C separately beneath it, and D joins the two together at the bottom to form a diamond shape. Of course the above statement can be extended to more than two classes like in our case: three classes ES_TEST_SUITE, ES_FIT, and ES_VERIFY inherit from ES_CONNECTION and ES_TEST_SUITE inherits from the three classes (see Fig. 5.18).

Eiffel deals with the diamond problem through the use of *select* and *rename* keywords, where the ancestor's methods to use in a descendant are explicitly specified. This allows the methods of the base class to be shared between its descendants or to even give each of them a separate copy of the base class.

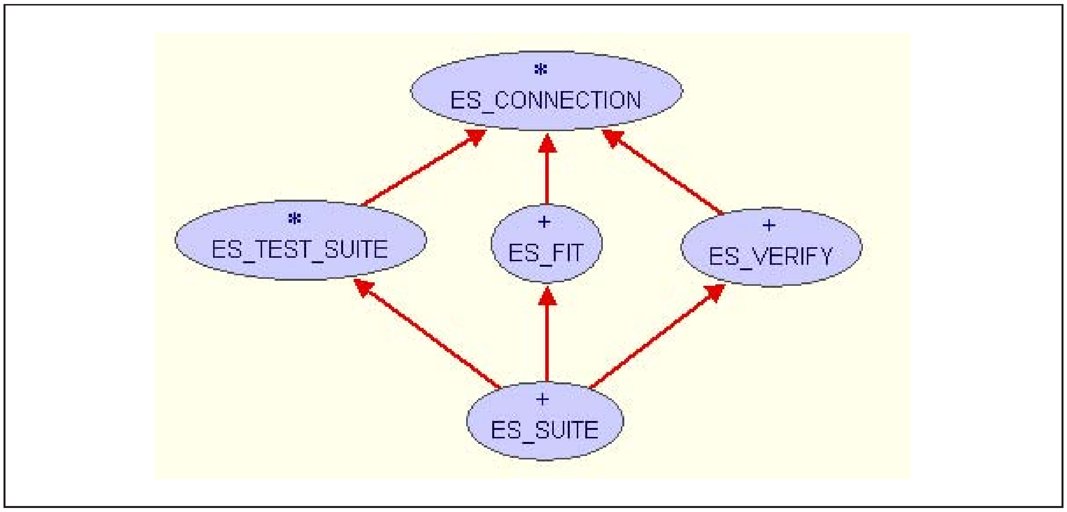


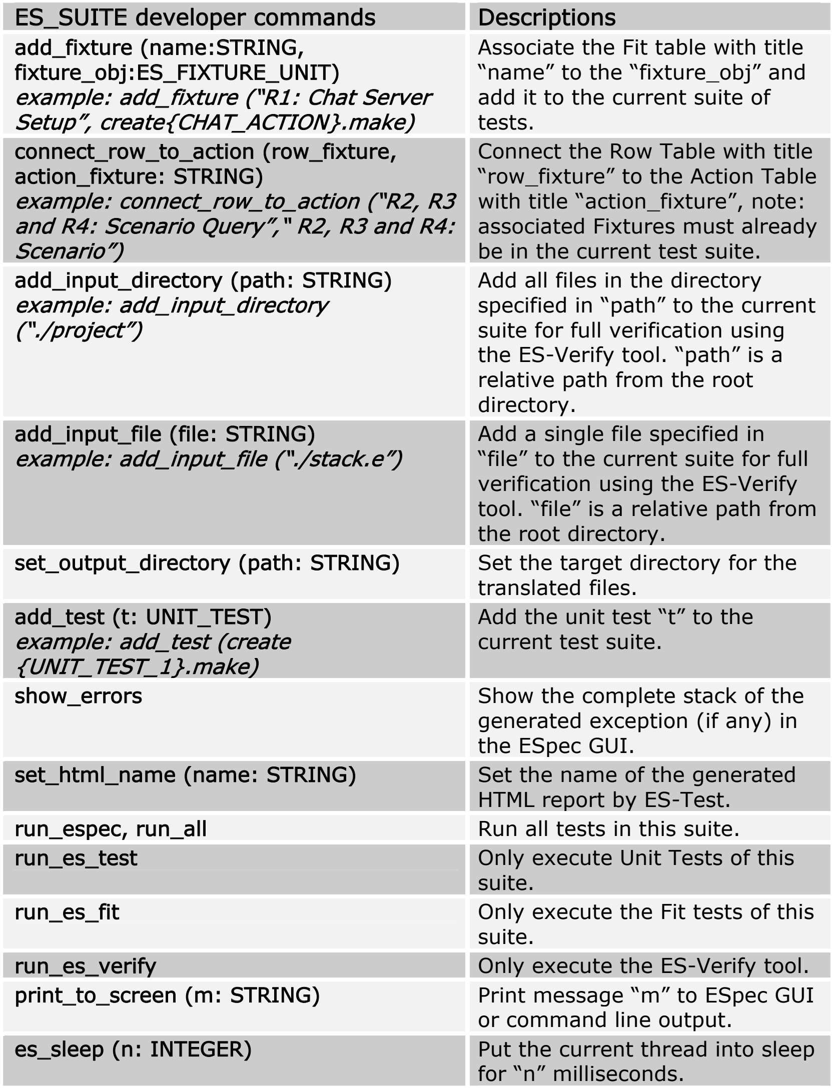
Figure 5.18: `ES_SUITE` shared interface

5.5 ESpec GUI

The purpose of ESpec GUI is to provide a convenient environment and graphical interface for users of ESpec library. ESpec GUI is written in Eiffel 6.0 and is cross-platform for Linux, UNIX and MS Windows. ESpec GUI is maintained in three flavors:

- ESpec Full (Research Edition) for Windows
- ESpec Student (Academic Edition) For Windows
- ESpec Student (Academic Edition) For Linux/Mac

We aimed to base our GUI design on well-known HCI principles [33]. It is targeted towards maximum usability. The basic principles of usability and the



ES_SUITE developer commands	Descriptions
add_fixture (name:STRING, fixture_obj:ES_FIXTURE_UNIT) *example: add_fixture (“R1: Chat Server Setup”, create{CHAT_ACTION}.make)*	Associate the Fit table with title “name” to the “fixture_obj” and add it to the current suite of tests.
connect_row_to_action (row_fixture, action_fixture: STRING) *example: connect_row_to_action (“R2, R3 and R4: Scenario Query”,” R2, R3 and R4: Scenario”)*	Connect the Row Table with title “row_fixture” to the Action Table with title “action_fixture”, note: associated Fixtures must already be in the current test suite.
add_input_directory (path: STRING) *example: add_input_directory (“./project”)*	Add all files in the directory specified in “path” to the current suite for full verification using the ES-Verify tool. “path” is a relative path from the root directory.
add_input_file (file: STRING) *example: add_input_file (“./stack.e”)*	Add a single file specified in “file” to the current suite for full verification using the ES-Verify tool. “file” is a relative path from the root directory.
set_output_directory (path: STRING)	Set the target directory for the translated files.
add_test (t: UNIT_TEST) *example: add_test (create {UNIT_TEST_1}.make)*	Add the unit test “t” to the current test suite.
show_errors	Show the complete stack of the generated exception (if any) in the ESpec GUI.
set_html_name (name: STRING)	Set the name of the generated HTML report by ES-Test.
run_espec, run_all	Run all tests in this suite.
run_es_test	Only execute Unit Tests of this suite.
run_es_fit	Only execute the Fit tests of this suite.
run_es_verify	Only execute the ES-Verify tool.
print_to_screen (m: STRING)	Print message “m” to ESpec GUI or command line output.
es_sleep (n: INTEGER)	Put the current thread into sleep for “n” milliseconds.

Table 5.6: ES_SUITE commands available to the developer

ways our application satisfies them are described below:

Predictability: The main functions are clearly displayed in the command panel on the right side of the screen. Standard Windows shortcuts are used whenever possible. The menus are organized in a familiar way. We try to keep them as standard and intuitive as possible.

Consistency: The function names are consistent throughout the application. Command names are consistent between the menus and the screen buttons.

Flexibility: Almost all the functionality can be activated by keyboard shortcuts.

Customizability: User may adjust such attributes as working directory, interaction mode, color scheme, font size, name of service directories, default ES-Clean settings, etc (see details in the user manual in Appendix D).

Observability: The current status is displayed on the screen at all times. The counters on the run panel are updated as tests run. This provides opportunity for the user to see the status incrementally. Test results, error messages, test cases and opened files are displayed to different displays in order to allow user to work with several documents in the same time and not to lose information.

Recoverability: The results of the tests that have already run are preserved, even if the testing is stopped. Error recovery is performed by displaying a hint to the user.

Responsiveness: The interface remains responsive even throughout long operations. If there is no need to continue running the rest of the tests, a button to stop the execution can be pressed.

The software developer may run all types of tests simultaneously under a single green/red bar via the *Run all Specs* button in ESpec tool (see Fig. 5.15). Alternatively, the developer can run a specific type of tests by invoking the associated button (i.e., *Run ES-Test*, *Run ES-Fit*, and *Run ES-Verify*). Test results for both types of tests are reported in the tool results window.

In addition, the user can *open* and *edit* HTML documents (for Fit requirements) using ESpec's internal HTML editor, perform the *ES-Clean* operation to remove compilation-generated files, archive the Eiffel source files (using *ES-Archive*), look up test cases in multiple source files, edit Eiffel source files, recompile existing Eiffel projects (freeze) and print test results and source code (see Appendix D for screenshots of the GUI).

The GUI code consists of two major clusters, which are the `ESPEC-GUI` cluster and the `ESPEC-LOGIC` cluster. They interoperate to various degrees to achieve the goals of the application. Fig. 5.19 shows the top-level diagram of the system.

5.5.1 The Business Logic cluster

The business logic class, `ES_LOGIC`, manages Eiffel project directories, and contains routines that execute all the testing tools (ES-Test, ES-Fit, ES-Verify) at the same time or each of them individually. This class also contains routines to execute ES-Archive, ES-Clean and Freeze utilities. The `ES_GLOBALS` class stores and

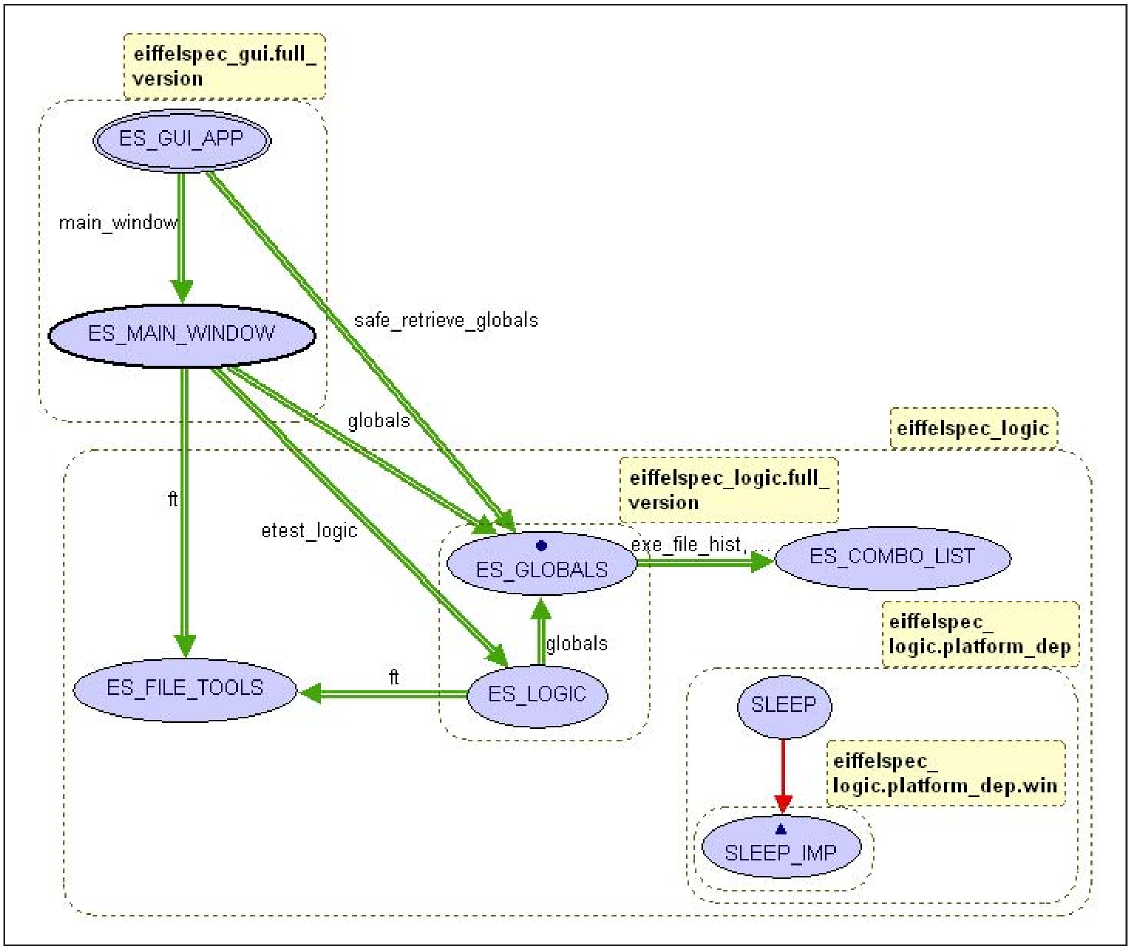


Figure 5.19: Cluster-level overview

provides access to global constants and shared variables.

The ES_FILE_TOOLS utility class provides simple file system related routines for the both the GUI and the logic classes. Fig. 5.20 shows the classes of the business logic cluster.

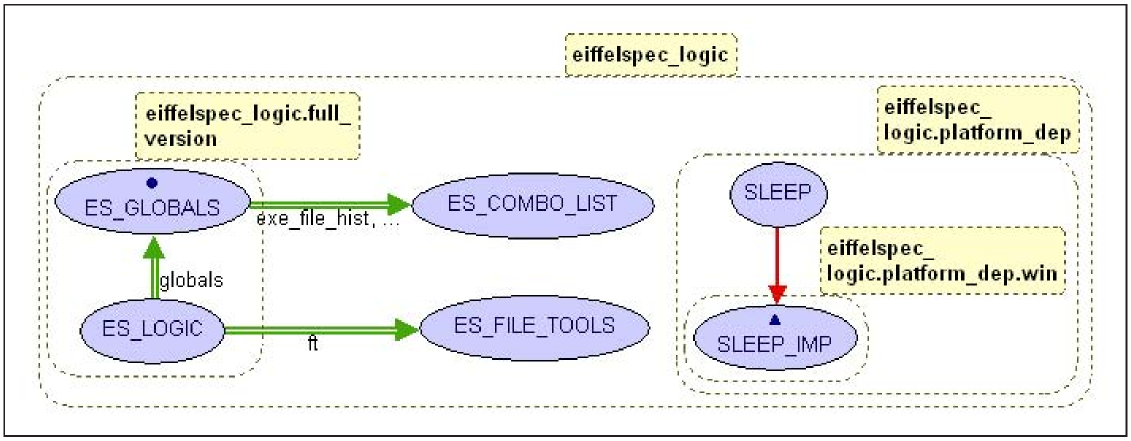


Figure 5.20: BON diagram of the ESpec logic cluster

5.5.2 The GUI cluster

The application's main window class, ES_MAIN_WINDOW, handles the interaction with the user of the system and uses the business logic classes to perform project maintenance operations and access test data. The GUI is modularized to make it easier to modify and maintain. Fig. 5.21 shows a high level overview of the contents of the main window.

The main window consists of four ES_PANELs. Each panel is responsible for a distinct part of the window's functionality. ES_MAIN_WINDOW plays the role of a *Mediator* between all the panels. This is important to allow for easy addition of functionality to the application. The custom panels, command buttons and blocks all augment Vision2 widgets with extra functionality (see Fig. 5.22).

The main command panel, ES_COMMAND_PANEL, consists of twelve buttons.

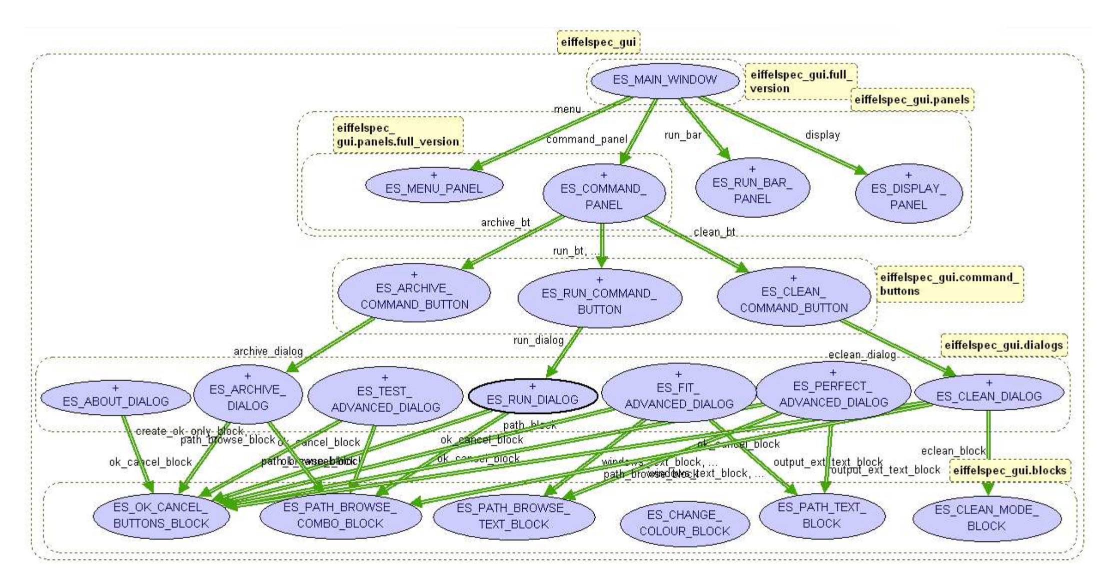


Figure 5.21: BON diagram of the ESpec gui cluster

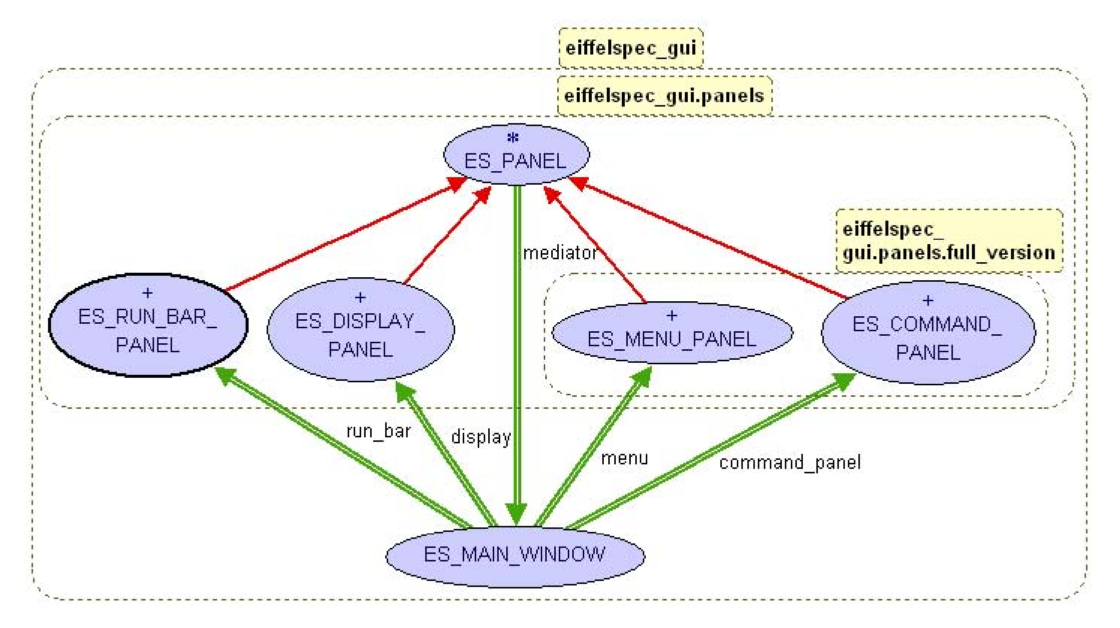


Figure 5.22: Main window as a mediator

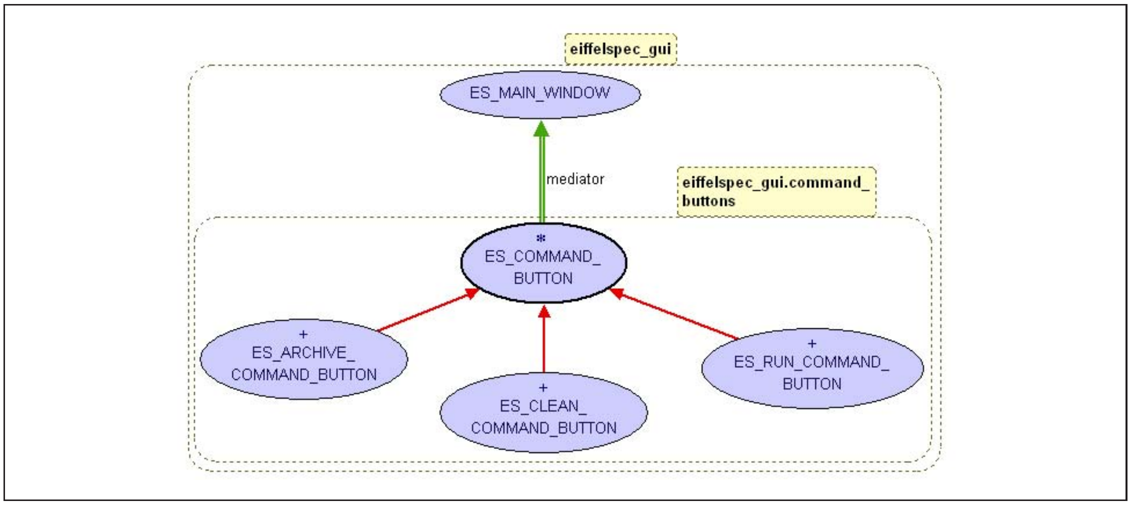


Figure 5.23: Command Buttons

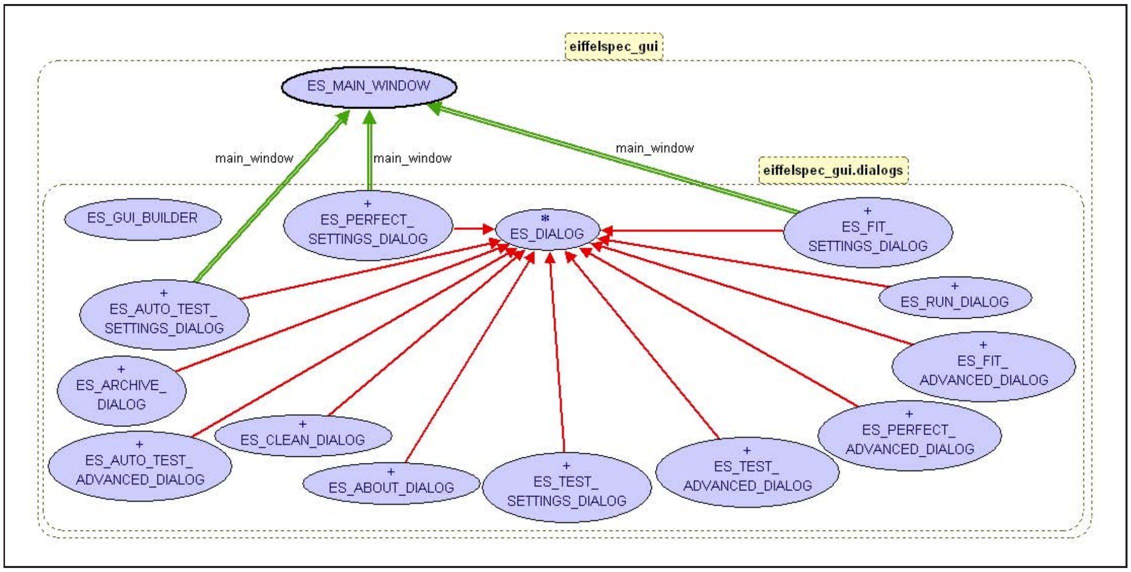


Figure 5.24: ESpec Dialogs

These buttons represented by implementations of the ES_COMMAND_BUTTON deferred class. The command buttons, for *Run all Specs*, *ES-Test*, *ES-Fit*, *ES-Verify*, *AutoTest* are handled by ES_RUN_COMMAND_BUTTON class and *ES-Archive* and *ES-*

Clean are handled by ES_ARCHIVE_COMMAND_BUTTON and ES_ECLEAN_COMMAND_BUTTON.

This architecture allows the business side of the application to focus entirely on business-related functionality. Fig. 5.23 shows the diagram associated with these buttons.

We build custom dialogs from dialog blocks, as can be seen in Figures 5.24 and 5.21 all the dialogs consist of blocks: such as path combo box plus browse button; or set of check boxes.

5.5.3 Summary of design patterns used

Our main window is a *Mediator* (see Fig. 5.22). It dispatches messages between the menu bar, the display and the command buttons, which have no direct knowledge of each other. This is done to lower the coupling between the components. ES_GLOBALS is a *Singleton* class. The nature of its data and functions is that it should exist only in one copy (like constants and shared variables). We decided to implement most of the GUI modules as subclasses of Vision2 widgets, allowing them to be inserted into the GUI structure directly (e.g. ES_COMMAND_BUTTONs). They maintain their GUI behaviour, such as responding to button clicks, but have additional functionality.

We decided to make the ES_GLOBALS class inherit from STORABLE so that we could easily maintain settings between uses of the applications. The application loads the structure from disk at start-up time and then saves it every time a parameter is changed.

5.6 Conclusions

In this chapter we explained the design decisions that were made in developing the ESpec library and the ESpec GUI. We showed the development of ES-Fit and our extensions to the original framework. We also showed the integrated interface architecture (`ES_SUITE`) that allows the developer to run various tools of the system individually or together for the purpose of testing.

6 Related Work

In Section 1.3 we described a rational software development as follows:

- Elicit and document the Requirements R of the customer in terms of the phenomena in the Problem Domain. Constraints of the Problem Domain are described by P.

- From the Requirements, derive Specifications S for the software code that must be developed.

- From the Specifications, derive a machine C (the code) that satisfies the Specifications.

This process was described as follows:

1. **Validation of Requirements:** $P \wedge S \rightarrow R$

2. **Verification of Specifications:** $C \rightarrow S$

3. **System Correctness:** From (1) and (2) conclude that: $P \wedge C \rightarrow R$

We describe requirements before proceeding to design because the problem to be solved should be specified before proceeding to solutions. We validate the requirements (formula 1) by showing that the specified solution (S) satisfies the customer requirements (R) in the problem domain (P). The Validation formula checks that we are developing the right product—the one desired by the customer as described by R.

Verification (formula 2) checks that the behaviour executed by the implemented code C satisfies the specification S. This formula checks that we are developing the product right.

Verification and Validation (V&V) have been pursued both formally and informally. Informal methods of V&V use, as their notations, English text and informal sketches, but also semi-formal notations such as UML which is used in Model Driven Development [82]. The advantage of English text is that customers understand it and can certify that what is being described is what they want. Fully formal methods are "mathematically-based techniques, often supported by reasoning tools, that can offer a rigorous and effective way to model, design and analyze computer systems" [42, 15, 27].

The Verification formula $C \rightarrow S$ is asserted in a way that is friendly to the idea of testable specifications and the use of formal methods. The formula asserts that a behaviour that satisfies the software implementation must also satisfy the design specification. Software developers (not customers) do verification and thus both the code and the specification can be described formally and even mathematically (customers are not expected to read these mathematical descriptions). Consequently, there is a vast literature on methods and tools for doing formal verification [27, 1, 83].

When it comes to Requirements Validation, the use of formal methods is much more sparse. There are, in fact, relatively few tools that check requirements (as opposed to specifications). There are many good commercial tools (such as DOORS [51]) for managing requirements. But these tools do not check that the design specification satisfies the customer requirements.

Methods and tools such as *i** and KAOS [90, 80] have been developed for requirements descriptions. These methods and tools address the early requirements stage in which goals must be elicited before the requirements of a product to be developed can be described. For example, a conference committee may have, as a first-sketch, the Goal that authors receive feedback within 4 weeks of the submission date. There are many ways to achieve this high-level goal. In order to achieve this goal, the Requirement may be to develop an online submission and refereeing application that helps programme chairs cope with the complexity of the refereeing process. This may include features such as management and monitoring of the programme committee and flexible facilities for management of the access of programme committee members and referees to papers that take into account conflicts of interests. Features might include facilities for automatic paper submission, paper assignment based on the preferences of PC members.[10] A Goal is likely to be more stable and last longer than the corresponding Requirement. The early phase aims to model and analyze stakeholder interests and goals and how they might be addressed, or compromised, by various system-and-environment alternatives. Requirements modeling techniques can be used to help deal with the knowledge and reasoning needed in this earlier phase of requirements engineering. While this is an important area of ongoing research, it is orthogonal to validating requirements in the context of this thesis which deals with the later stage in which the requirements (as opposed to just the goals) are better known.

The Problem Frames approach of Jackson [55] provides a framework for understanding the interaction between software and other system components. It emphasizes decomposing an end-to-end system requirement into a machine specification plus a set of assumptions about domains in the problem world. The standard approach does not provide the designer with a means for performing such a decomposition, apart from consulting a catalog of frame concern patterns. In [81], a more systematic method for transforming an end-to-end sys-

[10] See, for example, http://www.easychair.org/

tem requirement into a machine specification plus a set of domain properties is presented. The Problem Frames approach is extended to include a systematic way to transform an end-to-end system requirement into a machine specification. Given a Problem Frame description and an end-to-end requirement, a series of transformations turn the requirement into a specification and produce a set of breadcrumb assumptions about the problem world. The specification and breadcrumbs form a frame concern correctness argument for why the machine enforces the requirement. The Alloy tool [53] is used for this decomposition.

Our interest in this thesis has been specifying object oriented systems using Design by Contract using mathematical models (ML-Contracts). We will thus survey other methods and tools that support object-oriented Design by Contract. These tools have been developed only in the last decade or so.

Contributions of this thesis

Requirements are often described informally so that customers can read them. It is thus harder to come up with mechanically testable requirements than testable specifications. This is probably why there is much more research on formal Verification (formula 2) than formal Validation (formula 1). In this thesis we have focused on three aspects of Validation and Verification (V&V) as described in formulas (1), (2) and (3). The first aspect is that we insisted that in Validation we should strive to make requirements testable, and in Verification we should strive to make specifications testable. The second aspect was that testing should be supported by mechanized (possibly lightweight) formal methods tools. The third aspect was to integrate the various tools in a single toolset so that Validation of requirements and Verification of specifications are used in a coordinated way to check System Correctness. Because our method and tool (ESpec) deals with requirements and specifications it is wide-spectrum.

The Fit framework used in this thesis is a suitable way of achieving the goal of testable requirements in a way that can be fully integrated with testable specifications (ML-Contracts) so as to check system correctness as illustrated in the chat application case study in Chapter 4. The method and tool (ESpec) for integrated checks on testable requirements and specifications is the main contribution of this thesis. Violations of ML-Contracts is the integrative medium for checking requirements (the Fit tables) and the specifications. The specific contributions are detailed in Section 1.5.

6.1 Method and Tool comparison

Table 6.1 compares the most important methods and tools that support Design by Contract with respect to the ESpec tool developed in this thesis. The tools include JML [19], Perfect Developer [20], the KeY tool [2], and Spec# [9]. These

	ESpec	JML	Perfect Developer	Spec#	KeY
Testable Requirements	✓ (Fit framework)	✗	✗	✗	✗
Testable Specifications (executable mathematical models)	✓ (ML-Contracts)	✓ (Model Variables)	✗ (Perfect Language)	✗	✗ (OCL)
System Correctness (integrated testing of requirements and specifications)	✓	✗	✗	✗	✗

Table 6.1: Tool comparison in terms of three main characteristics

tools were designed to deal with the mechanical Verification of Specifications (not Requirements Validation) as shown in the table. All the tools (other than Spec#) use immutable mathematical models, but these mathematical models are not always executable. The advantage of executable mathematical models is that they can be used for runtime assertion checking in addition to formal verification using theorem proving techniques.

Each of these tools has advantages (and disadvantages) not shared by the other tools. Table 6.2 compares the tools (including ESpec) with respect to important features. We provide below more detailed comparisons including a brief overview of these tools.

As described in Chapter 2, the ES-Test component of the ESpec tool allows the developers to write and verify two kinds of specifications: ML-Contracts (i.e., Contracts written in ML) and Scenario Tests.

Scenario Tests provide the facilities for specifying and testing the collaboration between various modules of the system and ML-Contracts provide the ability describe a precise and complete high-level design specifications of the system. ES-Test can then execute the Scenario Tests which then have the amplifying effect of checking the ML-Contracts, while at the same time checking that the scenarios satisfy the specifications. Contract failures provide diagnostic feedback in Fit tables (requirements) as well as in the test report (specifications).

ES-Verify is the component of ESpec that uses theorem proving for verifying implementation correctness. The ES-Verify component translates Eiffel source code (annotated with ML-Contracts) into the specification language for the theorem prover and invokes the theorem prover to do the verification. [11] The current version works with a value semantics with ongoing work to extend it to reference semantics (see Appendix C).

[11] ES-Verify uses the Perfect Developer theorem prover which is one of the tools used in the comparison. In Perfect Developer, the software developer writes specifications in the special Perfect Specification language. The tool then generates Java or C++ executables based on the specifications. However, the contracts are not executable nor may a debugger be used at the Perfect language level. ES-Verify uses the theorem proving facilities at the Perfect specification language level, not the code generating facilities.

JML

The Java Modeling Language is a behavioral interface specification language for Java which extends Java with Design by Contract. JML uses some of the ideas from Eiffel, Larch [47] and the Refinement Calculus [5]. JML is used to specify the detailed design of Java classes and interfaces by adding contracts to Java source files. The aim of JML is to provide a specification language that is easy to use for Java programmers and that is supported by a wide range of tools for specification typechecking, runtime debugging, static analysis, and verification [19].

Like Eiffel, JML uses Java's expression syntax in assertions so it is easier for programmers to learn. However, unlike Eiffel, JML specifications must be written as comments in the Java source program and are ignored by the Java compiler `javac`. As a result, it is essential to use external JML tools (e.g., the JML syntax checker `jml` or the JML runtime assertion checker `jmlc`) to parse and debug the JML code.

In addition to supporting the Eiffel-style Design by Contract (such as precondition, postcondition and class invariants), JML introduces new keywords such as "signals" and "assignable". The "signals" keyword introduces the idea of *exceptional postconditions*, which allows JML developers to specify program behaviour in the case of exceptions.

The "assignable" keyword specifies the *frame conditions*. Frame conditions describe the properties that remain unchanged after a feature call. Frame conditions are essential for verification of code using the theorem provers. Although native Eiffel does not provide the capability to directly define such frame conditions, ESpec allows definition of frame constraints for the purpose of formal verification. We use the `pd_modify` declaration in the Eiffel code with its string argument passed as a list of attributes that the Eiffel feature may change.

A similarity between JML and ESpec is that both come with executable libraries that provide types that can be used for describing the specification mathematically. These (ML) libraries include such concepts as sets, lists, maps, sequences, and relations. They are similar to libraries of mathematical concepts found in VDM [58], Z [83], or OCL [88], with the difference that they are executable.

JML provides *Model Variables* which play the role of abstract values for abstract data types [24] allowing the developer to hide implementation details. The Eiffel language does not directly provide model declarations at the language level that guarantees side-effect free functions. However, the command-query separation rule is used instead to ensure that all queries may be used in contracts without affecting the state. Such an approach relies on the goodwill of programmers. However, ES-Verify can check that the resulting query implementations are indeed pure.

In JML, unit tests can be written and executed in Java's unit testing framework JUnit [41]; however, in order to get the contracts working, the JML code needs to be compiled using the JML compiler `jmlc`. Currently, JUnit and `jmlc` can be used together in the Eclipse IDE. The combination of JUnit and `jmlc` work like ES-Test. If during the execution of a unit test any contract violation is generated, it is reported in the Eclipse GUI (as assertion violations).

Another tool for unit testing of JML code is introduced in [23], called JMLUnit, which uses a JML runtime assertion checker to decide whether methods are working correctly, thus automating the writing of unit test oracles. These oracles can then be combined with hand-written test data. This tool is very similar to the random testing tool for Eiffel called AutoTest [25]. AutoTest allows the user to generate, compile and run tests on the push of a button and seamlessly integrates with existing manual unit tests. AutoTest relies on the contracts in the Eiffel code: it interprets contract violations as bugs. The research version of the ESpec tool include AutoTest as part of the package and is currently at the experimental stage.

For performing formal verification in JML, a number of third-party tools are available. Perhaps the most popular is the Extended Static Checker for Java (ES-

C/Java), and the later version ESC/Java2 [28]. ESC/Java2 is a programming tool that attempts to find common run-time errors in JML-annotated Java programs by static analysis of the program code and its formal annotations. ESC/Java2 is written with Java 1.4 and only runs on a Java 1.4 virtual machine; so new features offered by Java 1.5 (e.g., genericity) are not yet supported.

ESC/Java2 does not use models for the purpose of verification and only parses them, whereas ESpec's formal verification tool, ES-Verify, directly translates and verifies mathematical models included in the Eiffel code. These models not only provide a higher level abstractions than normal contracts, but also help to simplify the verification process.

ESC/Java2 uses the *Simplify* theorem prover for static verification of JML code [32]. *Simplify* works with integers and booleans primitive data types but not reals, characters and strings. By contrast, ESpec's theorem prover supports booleans, integers, reals, characters and strings.

ESC/Java2 provide precise feedback as to where errors occur. By contrast, the ESpec tool does not yet provide such precise feedback; however, the output HTML file produced by the theorem prover goes some way to providing feedback. The line number in the HTML points to the Eiffel feature having the same name or assertion tag, so that it is relatively easy to track back to where the problem was.

A number of other tools that help the verification process are available for JML (see [19] for an overview on JML tools). The LOOP/PVS tool is the most ambitious project to date for verifying Java code based on JML specifications. The authors of [56] describe the project as follows:

> Currently, the LOOP tool and ESC/Java 2 probably cover the largest subset of Java, and the LOOP tool probably supports the most complicated specification language.
>
> One distinguishing feature of the LOOP project is that it uses a shallow embedding of Java in [the theorem prover] PVS. This has both advantages and disadvantages.
>
> An advantage is that is has allowed us to give a completely formal proof of the soundness of all the programming logic we use, inside the theorem prover PVS ... A disadvantage of the use of a shallow embedding is that much of the reasoning takes places at the semantic level, rather than the syntactical level, which means that during the proof we have an uglier and, at least initially, less familiar syntax to deal with. Using the LOOP tool and PVS to verify programs requires a high level of expertise in the use of PVS, and an understanding of the way the semantics of Java and JML has been defined.
>
> A difference between LOOP and many of the others approaches ... [is that] the LOOP tool produces a single, big, proof obligation in PVS for every method, and then relies on the capabilities of PVS to reduce this proof obligation into ever smaller ones which we can ultimately prove. Most of the other tools already split up the proof obligation for a single method into smaller chunks (verification conditions) before feeding them to the theorem prover, for instance by using wp-calculi. A drawback of the LOOP approach is that the capabilities of theorem prover become a bottleneck sooner than in the other approaches.

Theorem proving tools are generally quite complex (a remark that applies equally to ES-Verify). The authors of [50] write as follows about the problems of integrating the various tools for JML.

> ... we describe our findings after integrating several tools based upon the Java Modeling Language (JML), a specification language used to annotate Java programs. The tools we consider are Daikon, ESC/-Java, JML runtime assertion checker, and LOOP/PVS tool. The first one generates specifications; the others are used to verify them. We find that for the first three it is worthwhile to combine them because this is relatively easy and it improves the specifications. Combining Daikon and the LOOP/PVS tool directly works in theory, but in practice it only works if the test suite is very good and hence it is not advisable...

Perfect Developer

Perfect Developer [30] is a tool with a formal specification language (the *Perfect Language*), a compiler for parsing the language, a theorem prover for verifying that implementations satisfy specifications and a code generator that transforms *Perfect* specifications to executable code (e.g. in Java). We use the abbreviation PD (Perfect Developer) to refer (somewhat imprecisely) to the method and any of its tools.

> More precisely, it [PD] is a specification language with an implementable subset identified as its programming language. The verifier is a custom-built theorem prover that collects and attempts to discharge proof obligations for the software it is presented with. The compiler accepts code written in the programming language and compiles it into equivalent Java, C++ or Ada95 code. Third party editors and UML modeling tools can be integrated into PD [20].

What is the motivation for PD? Currently industries use Programming languages (e.g. C++, Ada, Java) and, to a much lesser extent, Specification languages (e.g.

Z, VDM and B). These Programming and Specification languages are very different from each other, and it is often hard to relate a specification written in Z or VDM to a program written in a programming language. Also, the syntax of current specification languages is highly mathematical and difficult for programmers to learn. PD was designed to express both specifications and implementations of object-oriented software systems in a syntax familiar to programmers instead of the more mathematically inclined notations of Z and VDM. PD expressions, while being closer to programming notations, are nevertheless fully mathematical.

The PD specification language has the capability to deal with real numbers, characters, and strings in addition to the integer and boolean primitives. The PD specification language also has a mathematical library of generic sequences, sets, bags and maps, as well as predicate logic quantification [35].[12]

A limitation of PD is that it discourages reference semantics [30]. It is well-known that the presence of multiple references to a common object causes aliasing and makes sound and complete static verification problematic. Therefore, PD, unlike say Java and Eiffel, adopts a value semantics by default. In PD, if a reference semantics is adopted, then, roughly speaking, a `heap` declaration, e.g.

[12] The ESpec mathematical model library (ML) is translated into the PD specification language so that PD's theorem prover can be used to verify Eiffel code. See Appendix C for more on the ES-Verify component.

heap MyHeap, would be required. Escher Technologies Ltd. is in the process of developing better handling of a reference semantics.[13]

The theoretical foundations of PD are Floyd-Hoare logic and Dijkstra's weakest precondition calculus and it has the power of first-order predicate calculus, as well as a few higher-order constructs [29]. The prover generates verification conditions and aims for verifying the total correctness (termination and refinement satisfying specification) of the input code. It delivers either a proof, upon success in discharging all verification conditions, or otherwise a list of warnings, possibly accompanied by useful fix suggestions. Output from the prover can be in formats such as HTML or Tex. From an academic point of view, there is a lack of information about the inner workings of the PD theorem prover (as opposed to an interactive theorem-proving system such as *Isabelle* [17]). Ideally, the logical rules used in correctness proofs should be open for inspection so that independent trust can be established. However, the PD theorem prover does provide the complete proof, and thus the product is robust and suitable for engineering use [36].

PD-generated code (e.g. Java) is typically much longer and more complex than the original contract-based specification and is not intended to be read.

[13] Despite these limitations, we have adopted PD for automated deduction in our ES-Verify tool, and we are in the process of constructing a library of base Eiffel classes with a value semantics (see Appendix C) using the Eiffel **expanded** construct. As a future goal we have to expand our tool to handle verification of reference aliasing and inheritance.

The PD approach is useful if there is never a need to deal with the generated code. However, Perfect specifications are neither directly executable nor is there a debugger at the model level. The PD tool provides a basic GUI for doing the verification of the specification but does not integrate unit testing or run-time assertion checking into the tool.

Spec#

The Spec# programming language [8] is an extension (superset) of the Java-like object-oriented language C# with the addition of Design by Contract. It extends the type system to include non-null types and checked exceptions. Many errors in modern programs manifest themselves as null-dereference errors, suggesting the importance of a programming language providing the ability to discriminate between expressions that may evaluate to null and those that are not null [9]. This feature is not yet implemented in Eiffel, but it is in the new Eiffel ECMA specification [34].

Similar to Eiffel, the Spec# compiler is fully integrated into the VS.NET IDE (i.e., Visual Studio for the .NET platform), so there is no need for external tool support for compiling the source code (unlike JML). Contracts are written directly in the code (not in the form of comments) and are parsed and type checked by the Spec# compiler.

The Spec# compiler differs from Eiffel compiler in that it does not only produce executable code from a program written in the Spec# language, but also preserves all specifications into a language-independent format. Having the specifications available as a separate, compiled unit means program analysis and verification tools can consume the specifications without the need to either modify the Spec# compiler or to write a new source-language compiler [9].

Similar to JML and unlike Eiffel, Spec# supports exceptional postconditions which specify behaviour of the method when exceptions are thrown. Spec# allows a **throws** declaration to be combined with a postcondition that takes effect in the event that the exception is thrown.

Spec# supports a more sophisticated version of frame conditions than JML or ESpec. Method contracts can include **modifies** clauses, which restrict which pieces of the program state a method implementation is allowed to modify. Spec# also supports *wildcards* to specify entities for the modify clause (see [7]), which additionally address the problem of specifying the modification of state in subclasses.

Eiffel does not deal with the invariant problem introduced when re-entering a method (see [7]); however, Spec# makes it explicit when an object is in its steady state versus when it is exposed, which means the object is vulnerable to modifications. It introduces a block statement **expose** that explicitly indicates when an object's invariant may temporarily be broken.

The Spec# compiler statically enforces non-null types, emits run-time checks for method contracts and invariants. To exercise the contracts, the developer can use any of the testing tools available for .NET. *NUnit* [52] is the unit testing framework that has the majority of the market share. It was one of the first unit testing frameworks for the .NET platform. NUnit tests can be run several different ways. Like ES-Test, NUnit can be executed in from the GUI application or from the console's application or can be used as an integral part of VS.NET IDE as well.

Visual Studio Team System (VSTS) [46] is another testing platform from Microsoft, with unit testing as one of its testing types. VSTS supports other testing, such as functional and load testing. VSTS enjoys a close relationship with the VS.NET IDE. The IDE allows the developers to use a wizard to generate the unit tests from the code. VSTS includes *TestManager*, which is a GUI to allow the developer to select tests to run and to see the results of those tests. TestManager is similar to ES-Test with respect to regression testing. It helps to run all unit tests at once, or user can select which tests to run. Like ES-Test, VSTS supports debugging unit tests in the IDE, so that it's possible to set breakpoints and start debug run via unit test.

In terms of formal verification, Spec# provides a static program verifier. This component is fully integrated into VS.NET IDE and is called Boogie [6]. Boogie generates logical verification conditions from a Spec# program. Internally, it uses the *simplify* automatic theorem prover similar to ESC/Java2. Boogie has better capabilities than PD (and thus ES-Verify) with respect to references and aliasing. Boogie currently has some limitations (in contrast to ES-Verify). It does not support methods in contracts, quantifiers in loop invariants, genericity, real numbers, and loop variants. These limitations are likely to disappear over time. Perhaps the main difference between Spec# and ES-Verify is that Spec# does not yet support high-level mathematical libraries (ML) for describing program properties neither in run-time assertion checking nor in program verification.

KeY

KeY [2] is a GUI based tool that provides facilities for formal specification and verification of programs within a commercial platform for UML based software development. Like ESpec, KeY aims at integrating formal specification and verification of software into the software development process.

The target language of KeY tool is Java Card [22]. Java Card is a proper subset of Java, excluding certain features (like threads, cloning or dynamic class loading) and with a much reduced API. KeY allows the developer to use a combination of graphical UML diagrams [63], OCL specifications [88] (or alternatively,

	ESpec	JML	Perfect Developer	Spec#	KeY
Support for Design by Contract	✓	✓	✓	✓	✓
Executable contracts	✓	✓	✗	✓	✗
Debugging contracts	✓	✗	✗	✓	✗
Command line execution	✓	✓	✓	✓	✗
Support for mathematical models	✓	✓	✓	✗	✗
Executable mathematical models	✓	✓	✗	✗	✗
IDE integration	✗	✓	✗	✓	✓
Fully automatic theorem prover	✓	✓	✓	✓	✗
Verification of mathematical contracts and frame conditions	✓	✗	✓	✗	✗
Verification of genericity	✓	✗	✓	✗	✗
Verification of real numbers support	✓	✗	✓	✗	✗
Verification of inheritance	✗	✓	✓	✓	✓
Verification of aliasing	✗	✗	✗	✓	✓
Exceptional post condition	✗	✓	✗	✓	✓
Precise theorem prover feedback to the source code in an integrated GUI	✗	✓	✓	✓	✗
Seamless integration of Unit testing and theorem proving	✓	✗	✗	✗	✗

Table 6.2: Summary of comparisons

JML) to write the specification for the Java Card source code. OCL is poorly understood at this point and is undergoing changes. This makes it difficult to provide a calculus for reasoning about specifications in OCL. The KeY project dealt with this problem as follows [2, p10]:

> Although possible, there are good arguments against building such a calculus directly for OCL:
>
> - It is difficult and expensive to develop a theorem prover for a given formal language. OCL is a big language compared to logic languages (such as first-order logic) and, in contrast to them,

> proof search in OCL is not well understood. Moreover, OCL is frequently revised.
>
> - OCL was not designed with proof support in mind, and like UML it is independent of the implementation language. It does not know about concrete implementations of datatypes such as the integers. Before version 2.0, there was no way to specify initial states of classes. OCL is also not intended to express complex proof obligations that involve several invariants (see below).
>
> As a consequence, we take a "compilation" approach: OCL expressions are translated into formulae of first-order logic (FOL). OCL compilation circumvents the difficulties outlined above. It also makes KeY independent from OCL as the sole specification language: recently, JML emerged as a popular specification language used in many formal methods projects dealing with JAVA and JAVA CARD [19]. Replacing the OCL to FOL compiler with a JML front end enables the use of KeY with JML. A further major advantage of translating OCL and JML into FOL is that we do not need to define a dedicated formal semantics for these specification languages. Their semantics is implicitly defined by the translation into FOL, the latter having a standard semantics that is widely agreed upon. The translation approach works only if it is natural to represent a specification language by FOL. Admittedly, this is not the case for "vanilla" FOL as encountered in logic textbooks. Object types, undefined expressions, and predefined operators need to be added to the syntax, semantics, and calculus of FOL in order to allow a natural and adequate translation. None of these extensions to FOL is new, but surprisingly no tutorial treatment of this material accessible to non-specialists is available.

The KeY tool has a modeling component that consists of the CASE tool with extensions for formal specification. In ESpec, the specifications (ML-Contracts) are an executable part of the programming language and preconditions, postconditions and class invariants are language constructs. In KeY, OCL or JML specifications are annotations (comments) that are processed by the theorem prover, but they are not executable and are not supported by the Java debugger [2].

KeY is able to generate automatically constraints by using design pattern instantiations. For standard formulations predefined instantiations of design pattern exist and can be easily used. On the other hand the user is free to formulate any valid OCL statement without the assistance of KeY.

In terms of visual modeling, EiffelStudio also provides the similar facility for the users to model their code using the BON [77] diagrams. However, the benefit of the KeY tool over EiffelStudio is that it processes the UML diagrams and integrates them with OCL and JML specifications for the purpose of verification. KeY translates the UML model, the implementation (Java Card) and the specification (OCL, JML) into Java Card Dynamic Logic [12] proof obligations which are passed to the deduction component. Java Card Dynamic Logic is a program logic used by the KeY prover (deduction component). The deduction component is used to construct proofs for the generated Java Card Dynamic Logic proof obligations.

Unlike ES-Verify, the KeY prover is not fully automatic. The KeY prover is an interactive verification system combined with powerful automated deduction techniques. However, as stated in [37]:

> Java, UML, OCL, and CASE tools are familiar to software engineers and students alike, which helps in getting started. Nevertheless, KeY cannot be recommended for such target groups at present: the interactive prover and its interaction with the user are in their infancy

> and are inadequate for serious use. Moreover, OCL is not expressive enough to specify complex program behaviour. Considering that KeY is still in alpha stage, it seems to be worthwhile to reevaluate the system in a few years in order to see whether it lives up to expectations

It is undoubtedly the case that while KeY is relatively new and underdeveloped, it will in future become a serious verification tool given the research effort in place [2].

For a summary of comparison between ESpec and the various tools mentioned above see Table 6.2.

7 Future Work and Conclusions

7.1 Future work

ESpec is an on going project at the Software Engineering Lab at York University. It has been used in the Software Design course at York University since its first release in the Winter of 2005. ESpec is maintained under the GPL licence for public download (see `http://www.cse.yorku.ca/∼sel/espec/`).

As we showed in Table 6.2, ESpec does not precisely report the location of the theorem proving error in the source code. In our future project we are planing to improve this error reporting. Also there is an ongoing project to integrate ESpec tools directly into the Eiffel Studio IDE. During this experimental project, ES-Test component was successfully integrated into the IDE. This will allow Eiffel programmers to access ESpec tools directly from their working environment. We are also fully integrating the AutoTest tool into our ESpec.

Currently, working with the model libraries needs precise user interaction when debugging facilities are needed. Model libraries are very hard to debug especially when there is a contract violation. There is a need to develop advanced debugging tools specially fine tuned for these models. Finally, the goal is to support concurrent contracting via the SCOOP mechanism. This is an active and challenging area of current research [71].

7.2 Conclusions

In this thesis, we differentiated the *customer requirements* (in the problem domain) from *design specifications* (in the solution space). The design specifications are the artifact intermediate between implemented code and the customer requirements.

We argued that the customer requirements and design specifications should be testable and testable early in the design cycle leading to early detection of requirement and specification errors. We described a method (and the ESpec tool) for early requirement and specification descriptions and testing and showed how this technique allows us to detect bugs in both implementation and specification of the software product.

The core idea behind early testable requirements was that the problem is described before we search for a solution and the problem description drives the design.

We followed the single model principle, i.e., design specifications written using expressive mathematical models such as sets, bags, sequences and maps are contracts that are integrated into the program text itself. These tightly integrated specifications allowed us to detect inconsistencies between code, specifications and requirements as early as possible and during the lifetime of the code.

We described the customer requirements using Fit tables and specification violations (where they occur) were indicated in these Fit tables. We showed that the method does not depend on a particular code development methodology (e.g., Agile vs. Conventional) and whatever development methodology is preferred can be used.

A Appendix: Introduction to Eiffel

A.1 Eiffel

The tool support for this thesis is implemented in Eiffel language [67]. The language targeted by the tool is also Eiffel. For this reason, the following gives a brief overview of the language. Eiffel is a pure object oriented programming language. The main features of the language are: Static type system, Multiple inheritance, Constrained genericity and Design by Contract (DbC).

Fig. A.1 shows the Eiffel version of the Hello World example. The only class in this example is `HELLO_WORLD`. This class has only one feature: the procedure `make`. The implementation of this procedure prints the string *Hello world!*. In Eiffel, every class implicitly or explicitly inherits from the class `ANY`. The routine `print` is defined in class `ANY` for convenience reasons. The procedure `make` is also marked to be a creation procedure. Since it has no arguments it can serve as the program entry point. For more information about Eiffel programming language please refer to [67].

```
class
    HELLO_WORLD
create
    make
feature
    make
        do
            print ("Hello World!")
        end
end -- class HELLO_WORLD
```

Figure A.1: ”Hello World” program in Eiffel language

A.1.1 Eiffel Terminology

Eiffel has its own naming convention which often diverges from the conventions used in languages such as C++, Java or C#. Throughout this thesis, we use the Eiffel naming convention. A terminology-mapping from Eiffel to C++ [57] is provided in Table A.1. This table is based on the mapping from [57] and shows the entries relevant for this thesis.

A.1.2 Eiffel Agents

Agents are the key Eiffel technology used in this thesis. Agents are objects that represent operations; they are effectively closures from functional programming. Agents can be passed to different software elements, which can use the object to execute the operation whenever they want. Agents thus provide a way of separating the definition of a routine from its execution. Agents are also a way of combining high-level functions (operations acting on other operations) with static typing in Eiffel. The following is a simple example of an agent using Eiffel's GUI library EiffelVision. Suppose you want to add the routine `eval_state` to the list of event handlers that will be executed when a mouse click occurs on the widget `my_button`. To carry this out, we could execute Eiffel statement shown in agent expression A.1:

$$\texttt{my\_button.click\_actions.extend(}\textbf{agent}\ \texttt{eval\_state)} \tag{A.1}$$

The operation being added to the button is indicated by the **agent** keyword. The keyword distinguishes an operation call to `eval_state` from a binding of the operation to the button. In general, the argument to extend can be any agent expression. An agent expression will include an operation plus any context that the operation may need (e.g., arguments). *Predicate* agents are of significant use. These agents apply boolean-valued operations to collections. For example, agent expression A.2 applies the boolean-valued function `is_positive` to elements of the integer list `intlist`, and conjoins together the result. The question mark `?` indicates an open argument that is provided by iterating through the range arguments provided, i.e., it indicates an arbitrary element of `intlist`. In A.3, the boolean-valued function `perfect_cube` is applied to each element of the integer list intlist and the final result is the disjoins of all the results.

$$\texttt{intlist.for\_all (}\textbf{agent}\texttt{ is\_positive(?))} \tag{A.2}$$

$$\texttt{intlist.there\_exists(}\textbf{agent}\texttt{ perfect\_cube(?))} \tag{A.3}$$

Eiffel	**C++**
ancestor class	superclass
class	class
object	object
child class	derived class
parent class	base class
deferred feature	pure virtual function
deferred class	abstract base class
generic class	template
feature	function
function	virtual function
once function	n/a
descendent class	subclass
precursor	super()
attribute	data member
creation feature	constructor
assertion	assertion
require	n/a
ensure	n/a
invariant	n/a
variant	n/a
loop invariant	n/a
check	assert
x is do end	virtual void x() { }
x.f (expanded object)	x.f()
x.f (reference object)	x->f()
a = b	a == b
equal (a,b)	a == b
create x.make	y = new x
Result	return
violation	exception

Table A.1: Eiffel to C++ terminology mapping

B Appendix: Chat example source code

B.1 CHAT_SCENARIO_1

```
class CHAT_SCENARIO_1 inherit
  ES_ACTION_FIXTURE

create
  make

feature
  make is
    do
      bind ("[user]", agent set_user_name)
      bind ("Connect [user]", agent connect_user)
      bind ("[room]", agent set_room_name)
      bind ("[user] adds [room]", agent add_room)
      bind ("[user] makes [room] private", agent user_sets_room_to_private)
      bind ("[user list]", agent set_user_list)
      bind ("[user] allows [user list] in [room]", agent user_allows_list)
      bind ("Total number of users", agent num_server_users)
      bind ("Total number of rooms", agent num_server_rooms)
      -- second scenario, move
      bind ("move [user] to [room]", agent move_user_to_room)
    end

  temp_user: STRING

  temp_room: STRING

  temp_user_list: ARRAY[STRING]

  server: CHAT_SERVER
```

```
start (arg: STRING) is
    -- initalize the objects
  do
    if arg.is_equal ("Chat Server") then
      create server.make
    end
  end

set_user_name (a_name: STRING) is
  do
    temp_user := a_name
  end

set_room_name (a_name: STRING) is
  do
    temp_room := a_name
  end

num_server_rooms: INTEGER is
  do
    Result := server.room_count
  end

num_server_users: INTEGER is
  do
    Result := server.user_count
  end

connect_user is
  local
    a_user: CHAT_USER
  do
    create a_user.make (temp_user)
    server.connect (a_user)
  end

add_room is
  local
    a_room: CHAT_ROOM
    a_user: CHAT_USER
  do
    a_user := server.get_user (temp_user)
    a_room := a_user.create_room (temp_room)
    a_user.add_room (a_room)
  end

set_user_list (input: ARRAY[STRING]) is
```

```
    do
      temp_user_list := input
    end

  user_sets_room_to_private is
    local
      a_user: CHAT_USER
    do
      a_user := server.get_user (temp_user)
      a_user.set_private (temp_room)
    end

  user_allows_list is
    local
      a_user: CHAT_USER
      i: INTEGER
    do
      a_user := server.get_user (temp_user)
      from
        i := temp_user_list.lower
      until
        i > temp_user_list.upper
      loop
        a_user.allow_user (temp_user_list.item (i), temp_room)
        i := i + 1
      end

    end

  move_user_to_room is
    local
      a_user: CHAT_USER
    do
      a_user := server.get_user (temp_user)
      a_user.enter_room (temp_room)
    end
end
```

B.2 CHAT_SCENARIO_QUERY_1

```
class CHAT_SCENARIO_QUERY_1 inherit
  ES_ROW_FIXTURE[CHAT_ROOM]

create
  make

feature
  make is
      -- Bining of table headers to agents
    do
      bind ("Room name", agent get_room_name)
      bind ("Owner", agent get_room_owner)
      bind ("Occupants", agent get_room_occupants)
      bind ("Is public?", agent is_room_public)
      bind ("Permitted list", agent get_room_allowed_list)
    end


  get_room_name (a_room: CHAT_ROOM): STRING is
    do
      Result := a_room.name
    end

  get_room_owner (a_room: CHAT_ROOM): STRING is
    do
      Result := a_room.owner.user_name
    end

  get_room_occupants (a_room: CHAT_ROOM): ARRAY[STRING] is
    local
      loc: INTEGER
    do
      from
        loc := a_room.occupants.index
        a_room.occupants.start
        create Result.make (0, -1)
      until
        a_room.occupants.after
      loop
        Result.force (a_room.occupants.item.user_name, Result.count)
        a_room.occupants.forth
      end
      a_room.occupants.go_i_th (loc)
    end
  get_user_name (a_user: CHAT_USER): STRING is
    do
      Result := a_user.user_name
    end

  get_user_room (a_user: CHAT_USER): STRING is
    do
      Result := a_user.room.name
    end

  is_room_public (a_room: CHAT_ROOM): BOOLEAN is
    do
      Result := not a_room.is_private
    end

  get_room_allowed_list (a_room: CHAT_ROOM): ARRAY[STRING] is
    do
      Result := a_room.allowed_list
    end


  query (a_name: STRING): LINKED_LIST [CHAT_ROOM]
      -- argument ‘a_name’ is not used in this example
      -- LIST will contain ‘Lobby, Technical Support’
    local
      chat_scenario1: CHAT_SCENARIO_1
      chat_server: CHAT_SERVER
    do
      chat_scenario1 ?= connected_to
      check chat_scenario1 /= Void end
      chat_server := chat_scenario1.server
      Result ?= chat_server.rooms.deep_twin
    end
end
```

B.3 CHAT_TEST_1

```
class CHAT_TEST1 inherit
  ES_TEST

create
  make

feature -- Add cases

  make is
    do
      add_boolean_case (agent test_server_creation)
      add_boolean_case (agent test_user_creation)
      add_boolean_case (agent test_set_user_server)
      add_boolean_case (agent test_room_creation)
      add_boolean_case (agent test_set_room_server)
      add_boolean_case (agent user_connects_server)
      add_boolean_case (agent test_scenario_1)

      add_violation_case (agent conneting_same_user_twice)
      add_violation_case (agent connecting_two_users_same_name)
    end

feature -- Boolean Cases

  test_scenario_1: BOOLEAN is
    local
      server: CHAT_SERVER
      mike, anna: CHAT_USER
      mike_room: CHAT_ROOM
      users: LIST[CHAT_USER]
      rooms: LIST[CHAT_ROOM]
    do
      -- create the chat server and check it
      create server.make
      users := server.users
      rooms := server.rooms
      check server.user_count = 1 end
      check server.room_count = 1 end

      -- create 2 users Mike and Anna and connect them to the server
      create mike.make ("Mike")
      create anna.make ("Anna")
      server.connect (mike)
      server.connect (anna)
      check server.user_count = 3 and server.room_count = 1 end
      check mike.room = server.lobby and anna.room = server.lobby end
      check users.has(mike) and users.has(anna) end

      -- Mike creates a room ‘‘Technical Support"
      mike_room := mike.create_room ("Technical Support")
      mike.add_room (mike_room)
      check server.room_count = 2 end
      check not mike_room.is_private end
      check rooms.has(mike_room) end

      -- Mike changes the status of his room to private
      mike.set_private ("Technical Support")
      check mike_room.is_private end
      check not server.is_allowed (anna, "Technical Support") end

      -- Mike allows Anna to join the Technical Support room
      mike.allow_user ("Anna", "Technical Support")
      check server.is_allowed (anna, "Technical Support") end
      Result := True
    end

  test_server_creation: BOOLEAN is
    local
      chat_server: CHAT_SERVER
      Lobby_name, Admin_name: STRING
    do
      comment ("Create chat server")
      Lobby_name := "Lobby"
      Admin_name := "Admin"
      create chat_server.make
      Result := chat_server.is_active
      Result := Result and chat_server.rooms.count = 1
      Result := Result and chat_server.lobby.name.is_equal (Lobby_name)
      Result := Result and chat_server.users.count = 1
      Result := Result and (chat_server.admin /= void and
               chat_server.admin.user_name.is_equal (Admin_name) and
               chat_server.admin.server = chat_server)
      Result := Result and chat_server.rooms.has (chat_server.lobby)
      Result := Result and chat_server.users.has (chat_server.admin)
      Result := Result and chat_server.lobby.occupants.has (chat_server.admin)
      Result := Result and chat_server.lobby.owner = chat_server.admin
    end

  test_user_creation: BOOLEAN is
    local
      a_user: CHAT_USER
    do
```

```
    comment ("Create a user")
    create a_user.make ("Mike")
    Result := a_user.user_name.is_equal ("Mike")
    Result := Result and a_user.server = Void
    Result := Result and a_user.room = Void
    Result := Result and a_user.owned.is_empty
  end

test_set_user_server: BOOLEAN is
  local
    a_user: CHAT_USER
    a_server: CHAT_SERVER
  do
    comment ("Set chat server for a user")
    create a_user.make ("Mike")
    create a_server.make
    Result := a_user.server = Void
    a_user.set_server (a_server)
    Result := Result and a_user.server = a_server
  end

test_room_creation: BOOLEAN is
  local
    a_user: CHAT_USER
    a_room: CHAT_ROOM
  do
    comment ("Create a room")
    create a_user.make ("Mike")
    a_room := a_user.create_room ("Mike’s room")
    Result := a_room.owner = a_user
    Result := Result and a_room.name.is_equal ("Mike’s room")
    Result := Result and a_user.owned.has (a_room)
    Result := Result and a_room.occupants.is_empty
  end

test_set_room_server: BOOLEAN is
  local
    a_user: CHAT_USER
    a_room: CHAT_ROOM
    a_server: CHAT_SERVER
  do
    comment ("Set room server")
    create a_user.make ("Mike")
    create a_server.make
    a_room := a_user.create_room ("A")
    Result := a_room.server = Void
    a_room.set_server (a_server)
```

```
      Result := Result and a_room.server = a_server
    end

  user_connects_server: BOOLEAN is
      -- user connects to a chat server
    local
      server: CHAT_SERVER
      mike: CHAT_USER
    do
      comment ("User connects to a chat server")
      create server.make
      create mike.make ("Mike")
      server.connect (mike)
      Result := mike.server = server
      Result := Result and mike.room = server.lobby
      Result := Result and server.users.has (mike)
      Result := Result and server.user_count = 2
      Result := Result and server.lobby.has_username ("Mike")
      Result := Result and server.has_user ("Mike")
    end

  conneting_same_user_twice is
    local
      server: CHAT_SERVER
      user1: CHAT_USER
    do
      comment ("conneting same user twice")
      create server.make
      create user1.make ("user1")
      server.connect (user1)
      server.connect (user1)
    end

  connecting_two_users_same_name is
    local
      server: CHAT_SERVER
      user1, user2: CHAT_USER
    do
      comment ("connecting_two_users_same_name")
      create server.make
      create user1.make ("user1")
      create user2.make ("user1")
      server.connect (user1)
      server.connect (user2)
    end
end
```

B.4 CHAT_TEST_2

```
class CHAT_TEST2 inherit
  ES_TEST
create
  make

feature -- Defining tests

  make is
    do
      add_violation_case_with_tag ("not_already_connected", agent
          conneting_same_user_twice)
      add_violation_case_with_tag ("user_names_unique", agent
          connecting_two_users_same_name)
      add_violation_case_with_tag ("user_connected", agent
          adding_a_room_without_connecting)
      add_violation_case_with_tag ("user_is_owner", agent adding_a_non_owned_room)
      add_violation_case_with_tag ("room_is_new", agent adding_an_existing_room)
      add_violation_case_with_tag ("room_is_new", agent adding_the_same_room_twice)
      add_violation_case_with_tag ("user_not_already_in", agent
          entering_a_room_twice)
      add_violation_case_with_tag ("user_allowed", agent entering_a_private_room)
      add_violation_case_with_tag ("user_allowed", agent switching_rooms_to_private)
      add_violation_case_with_tag ("not_already_allowed", agent allowing_owner)
      add_violation_case_with_tag ("user_is_owner", agent
          making_non_owned_rooms_private)
      add_violation_case_with_tag ("user_is_owner", agent
          changing_allowed_list_of_non_owned_rooms)
      add_violation_case_with_tag ("room_exists", agent removing_a_non_existing_room
          )
      add_violation_case_with_tag ("user_is_owner", agent removing_a_non_owned_room)
      add_violation_case_with_tag ("user_is_owner", agent removing_lobby)
      add_violation_case_with_tag ("already_connected", agent
          disconnecting_without_being_connected )
    end


feature -- Agents

  conneting_same_user_twice is
    do
      comment ("conneting same user twice")
      create server.make
      create user1.make ("user1")
      user1.connect (server)
      user1.connect (server)
    end

  connecting_two_users_same_name is
    do
      comment ("connecting_two_users_same_name")
      create server.make
      create user1.make ("user1")
      create user2.make ("user1")
      user1.connect (server)
      user2.connect (server)
    end

  adding_a_room_without_connecting is
    do
      comment ("adding_a_room_without_connecting")
      create server.make
      create user1.make ("user1")
      room1 := user1.create_room ("room1")
      user1.add_room (room1)
    end

  adding_a_non_owned_room is
    do
      comment ("adding_a_non_owned_room")
      create server.make
      create user1.make ("user1")
      create user2.make ("user2")
      room1 := user1.create_room ("room1")
      room2 := user2.create_room ("room2")
      user1.connect (server)
      user2.connect (server)
      user1.add_room (room2)
    end

  adding_a_room_same_as_username is
    do
      comment ("adding_a_room_same_as_username")
      create server.make
      create user1.make ("user1")
      create user2.make ("user2")
      room1 := user1.create_room ("room1")
      room2 := user2.create_room ("user1")
      user1.connect (server)
      user2.connect (server)
      user1.add_room (room1)
      user2.add_room (room2)
    end
```

```
adding_an_existing_room is
  do
    comment ("adding_an_existing_room_name")
    create server.make
    create user1.make ("user1")
    create user2.make ("user2")
    room1 := user1.create_room ("room1")
    room2 := user2.create_room ("room1")
    user1.connect (server)
    user2.connect (server)
    user1.add_room (room1)
    user2.add_room (room2)
  end

adding_the_same_room_twice is
  do
    comment ("adding_the_same_room_twice")
    create server.make
    create user1.make ("user1")
    create user2.make ("user2")
    room1 := user1.create_room ("room1")
    user1.connect (server)
    user2.connect (server)
    user1.add_room (room1)
    user1.add_room (room1)
  end

entering_a_room_twice is
  do
    comment ("entering_a_room_twice")
    create server.make
    create user1.make ("user1")
    create user2.make ("user2")
    room1 := user1.create_room ("room1")
    room2 := user2.create_room ("room2")
    user1.connect (server)
    user2.connect (server)
    user1.add_room (room1)
    user2.add_room (room2)
    user1.enter_room (room2.name)
    user1.enter_room (room2.name)
  end

entering_a_private_room is
  do
    comment ("entering_a_private_room")
    create server.make
    create user1.make ("user1")
    create user2.make ("user2")
    room1 := user1.create_room ("room1")
    user1.connect (server)
    user2.connect (server)
    user1.add_room (room1)
    user1.set_private (room1.name)
    user1.enter_room (room1.name)
    user2.enter_room (room1.name)
  end

switching_rooms_to_private is
  do
    comment ("switching_rooms_to_private")
    create server.make
    create user1.make ("user1")
    create user2.make ("user2")
    room1 := user1.create_room ("room1")
    room2 := user2.create_room ("room2")
    user1.connect (server)
    user2.connect (server)
    user1.add_room (room1)
    user2.add_room (room2)
    user1.set_private (room1.name)
    user2.enter_room (room2.name)
    user2.enter_room (room1.name)
  end

allowing_owner is
  do
    comment ("allowing_owner")
    create server.make
    create user1.make ("user1")
    room1 := user1.create_room ("room1")
    user1.connect (server)
    user1.add_room (room1)
    user1.set_private (room1.name)
    user1.allow_user (user1.user_name, room1.name)
  end

making_non_owned_rooms_private is
  do
    comment ("making_non_owned_rooms_private")
    create server.make
    create user1.make ("user1")
    create user2.make ("user2")
    room1 := user1.create_room ("room1")
    room2 := user2.create_room ("room2")
    user1.connect (server)
    user2.connect (server)
    user1.add_room (room1)
    user2.add_room (room2)
    user2.set_private (room1.name)
  end

changing_allowed_list_of_non_owned_rooms is
  do
    comment ("changing_allowed_list_of_non_owned_rooms")
    create server.make
    create user1.make ("user1")
    create user2.make ("user2")
    room1 := user1.create_room ("room1")
    room2 := user2.create_room ("room2")
    user1.connect (server)
    user2.connect (server)
    user1.add_room (room1)
    user2.add_room (room2)
    user2.allow_user (user2.user_name, room1.name)
  end

removing_a_non_existing_room is
  do
    comment ("removing_a_non_existing_room")
    create server.make
    create user1.make ("user1")
    create user2.make ("user2")
    room1 := user1.create_room ("room1")
    room2 := user2.create_room ("room2")
    user1.connect (server)
    user2.connect (server)
    user1.remove_room (room1)
  end

removing_a_non_owned_room is
  do
    comment ("removing_a_non_owned_room")
    create server.make
    create user1.make ("user1")
    create user2.make ("user2")
    room1 := user1.create_room ("room1")
    room2 := user2.create_room ("room2")
    user1.connect (server)
    user2.connect (server)
```

```
      user1.add_room (room1)
      user2.add_room (room2)
      user1.remove_room (room2)
    end

  removing_lobby is
    do
      comment ("removing_lobby")
      create server.make
      create user1.make ("user1")
      user1.connect (server)
      user1.remove_room (server.lobby)
    end

  disconnecting_without_being_connected is
    do
      comment ("disconnecting_without_being_connected")
      create server.make
      create user1.make ("user1")
      user1.disconnect
    end


  user1, user2, user3, user4, user5: CHAT_USER
  room1, room2, room3, room4, room5: CHAT_ROOM
  server: CHAT_SERVER
end
```

B.5 CHAT_ROOM

```
class
  CHAT_ROOM
create
  make

feature {CHAT_SERVER, UNIT_TEST, ES_FIXTURE_UNIT} -- access private
  occupants: LIST [CHAT_USER] -- list of occupants in this chat room
  owner: CHAT_USER     -- user who owns the room
  server: CHAT_SERVER     -- server associated with the room

feature -- access
  allowed_list: ARRAY [STRING] -- list of allowed people in the room
  is_private: BOOLEAN     -- status of the room (public or private)
  name: STRING      -- name of the room

feature -- Room actions

  make (a_name: STRING_8; a_user: CHAT_USER) is
      -- creates a chat room with name 'a_name'
      -- and owner 'a_user'
    require
      a_name_non_void: a_name /= void
      a_name_non_empty: not a_name.is_empty
      a_user_non_void: a_user /= void
    do
      name := a_name
      occupants := create {LINKED_LIST[CHAT_USER]}.make
      owner := a_user
      create allowed_list.make (0, -1)
      allowed_list.compare_objects
      allowed_list.force (a_user.user_name, allowed_list.count)
    ensure
      name_set: name.is_equal (a_name)
      occupants_empty: occupant_model.is_empty
      owner_assigned: owner = a_user
      allowed_list_created: allowed_list.count = 1
    end

feature -- query
  has_username (a_name: STRING): BOOLEAN is
      -- returns true if user with 'a_name' is in this room
    require
      a_name_non_void: a_name /= Void
    local
      loc: INTEGER
    do
      from
        loc := occupants.index
        occupants.start
      until
        occupants.after or Result
      loop
        if occupants.item.user_name.is_equal (a_name) then
          Result := true
        else
          occupants.forth
        end
      end

      occupants.go_i_th (loc)
    ensure
      Result = occupant_model.there_exists (agent user_with_name (?, a_name))
    end

  is_user_allowed (a_user: STRING): BOOLEAN is
      -- true if 'a_user' is allowed
    require
      a_user_non_void: a_user /= void
    do
      Result := allowed_list.has (a_user)
    end

  is_owner (a_user: CHAT_USER): BOOLEAN is
    require
      a_user_non_void: a_user /= void
    do
      Result := (a_user = owner)
    ensure
      result_ok: Result = (a_user = owner)
    end

  is_allowed (a_user: CHAT_USER): BOOLEAN is
      -- true if 'a_user' is allowed to enter the current room
    require
      a_user_non_void: a_user /= void
    do
      Result := not is_private or (is_private and allowed_list.has (a_user.
          user_name))
    end

  is_in_allowed_list (a_user: CHAT_USER): BOOLEAN is
      -- true if 'a_user' is allowed to enter the current room
    require
      a_user_non_void: a_user /= void
    do
      Result := allowed_list.has (a_user.user_name)
    end

feature {CHAT_SERVER, UNIT_TEST, ES_FIXTURE_UNIT} -- server access
  set_server (a_server: CHAT_SERVER) is
      -- set the server
    require
      a_server_non_void: a_server /= void
    do
      server := a_server
    ensure
      server = a_server
    end

  remove_user (a_user: CHAT_USER) is
      -- removes the ’a_user’ from the chat room
    require
      a_user_non_void: a_user /= void
      user_exists: occupant_model.has (a_user)
    do
      occupants.start
      occupants.search (a_user)
      occupants.remove
      occupants.start
    ensure
      user_removed: occupant_model.extended_by (a_user) |=| old occupant_model
    end

  add_user (a_user: CHAT_USER) is
      -- adds ’a_user’ to the current room
    require
      a_user_non_void: a_user /= Void
      not_already_in: not occupant_model.has (a_user)
      a_user_must_be_allowed: is_allowed (a_user)
    do
      occupants.force (a_user)
      a_user.set_room (Current)
    ensure
      user_added: occupant_model |=| (old occupant_model).extended_by (a_user)
    end

  set_allowed_list (list: ARRAY [STRING]) is
    require
      list_not_void: list /= void
    do
      allowed_list := list
      allowed_list.compare_objects
    ensure
      allowed_list = list
    end

  set_private is
      -- set room to be of type private
    require
      not_already_private: not is_private
    do
      is_private := true
    ensure
      is_private
    end

  set_public is
      -- set room to be of type public
    require
      not_already_private: is_private
    do
      is_private := false
    ensure
      not is_private
    end

  has_user (a_user: CHAT_USER): BOOLEAN is
    require
      a_user_non_void: a_user /= void
    do
      Result := occupants.has (a_user)
    ensure
      result_ok: Result = occupant_model.has (a_user)
    end

  allow_user (a_user: CHAT_USER) is
      -- allow 'a_user' to access this room
    require
      a_user_non_void: a_user /= Void
      not_already_allowed: not is_in_allowed_list (a_user)
    do
      allowed_list.force (a_user.user_name, allowed_list.count)
    ensure
      user_allowed: is_allowed (a_user)
    end
```

```
feature {NONE} -- agent

  user_with_name (a_user: CHAT_USER; a_name: STRING): BOOLEAN is
      -- true if 'a_user's name is 'a_name'
    require
      a_user_non_void: a_user /= void
      a_name_non_void: a_name /= void
    do
      Result := a_user.user_name.is_equal (a_name)
    ensure
      Result = a_user.user_name.is_equal (a_name)
    end

  must_be_allowed (a_user: CHAT_USER): BOOLEAN is
      -- true if 'a_user' is allowed to enter the current room
    require
      a_user_non_void: a_user /= void
    do
      Result := allowed_list.has (a_user.user_name)
    end

feature -- Model
  occupant_model: ML_SET [CHAT_USER] is
      -- model of the occupants in this room
    local
      loc: INTEGER
    do
      from
        create Result.make
        loc := occupants.index
        if occupants /= void then
          occupants.start
        end
      until
        occupants = void or occupants.after
      loop
        Result := Result.extended_by (occupants.item)
        occupants.forth
      end
      occupants.go_i_th (loc)
    end

invariant
  all_users_authorized: is_private implies occupant_model.for_all (agent
      must_be_allowed (?))
  owner_is_always_allowed: is_allowed (owner)
end
```

B.6 CHAT_USER

```
class CHAT_USER

create
  make

feature -- attributes
  user_name: STRING  -- username (login name) of the current user

feature {CHAT_ROOM, CHAT_SERVER, UNIT_TEST, ES_FIXTURE_UNIT} -- access
  room: CHAT_ROOM  -- room (location) of the current user
  server: CHAT_SERVER -- the server that current user is connected to
  owned: LINKED_LIST[CHAT_ROOM] -- list of rooms that is owned by current user

feature -- model
  owned_model: ML_SET[CHAT_ROOM] is
      -- returns a model representing the list of rooms this user owns
    local
      loc: INTEGER
    do
      from
        create Result.make
        if owned /= Void then
          loc := owned.index
          owned.start
        end
      until
        owned = Void or owned.after
      loop
        Result := Result.extended_by (owned.item)
        owned.forth
      end

      if owned /= Void then
        owned.go_i_th (loc)
      end

    end

feature {CHAT_SERVER, UNIT_TEST, ES_FIXTURE_UNIT} -- create / connect user
  make (a_user_name: STRING) is
      -- create a chat user with user name 'a_user_name'
    require
      a_name_non_void: a_user_name /= void
      a_name_non_empty: not a_user_name.is_empty
    do
```

```
      server := Void
      room := Void
      create owned.make
      user_name := a_user_name
    ensure
      name_is_set: user_name.is_equal (a_user_name)
      chat_server_void: server = Void
      current_room_void: room = Void
      owned_list_created: owned_model.is_empty
    end

  connect (a_server: CHAT_SERVER) is
      -- user connects to ‘a_server’
      -- should not have already connected
      -- chat server may accept/reject the user
    require
      not_already_connected: not is_connected
      server_is_active: a_server /= Void and a_server.is_active
    do
      a_server.connect (Current)
    ensure
      user_connected: is_connected
      owned_model_unchanged: owned_model |=| old owned_model
    end

  disconnect is
      -- disconnect the user from the chat server
    require
      already_connected: is_connected
    do
      server.disconnect (Current)
      create owned.make -- clear ownerships
      server := Void
      room := Void
    ensure
      not_connected: not is_connected
      connected_chat_server_void: server = Void
      not_in_room: room = Void
      server_not_has_user: not (old server).has_user (Current.user_name)
      model_empty: owned_model.is_empty
    end
feature -- query
  is_connected: BOOLEAN is
      -- true if user connected to a server
    do
      if server /= Void then
        Result := server.has_user (Current.user_name)
      end
    end

  is_in_room (a_room: STRING): BOOLEAN is
      -- true if user is in room a_room
    require
      a_room_non_void: a_room /= Void
    do
      Result := room.name.is_equal (a_room)
    ensure
      result_correct: Result = room.name.is_equal (a_room)
    end

  is_room_owned (a_room: STRING): BOOLEAN is
      -- true if ’a_room’ is owned by this user
    require
      a_room_non_void: a_room /= Void
    local
      loc: INTEGER
    do
      from
        loc := owned.index
        owned.start
      until
        owned.after or Result
      loop
        if owned.item.name.is_equal (a_room) then
          Result := true
        end
        owned.forth
      end
      owned.go_i_th (loc)
    end


feature {CHAT_SERVER, CHAT_ROOM, UNIT_TEST, ES_FIXTURE_UNIT} -- user actions
  set_server (a_server: CHAT_SERVER) is
      -- set the current of server to ’a_server’
    require
      a_server_non_void: a_server /= void
    do
      server := a_server
    ensure
      server_set: server = a_server
      owned_unchaged: owned_model |=| old owned_model
    end
```

```
set_room (a_room: CHAT_ROOM) is
    -- set the current room of this user to be ‘a_room’
  require
    a_room_non_void: a_room /= Void
    a_room_on_server: server.has_room (a_room.name)
  do
    room := a_room
  ensure
    room_set: room = a_room
    owned_unchaged: owned_model |=| old owned_model
  end

create_room (a_name: STRING): CHAT_ROOM is
    -- create a room with name ’a_name’
    -- user becomes the owner of the create room
  require
    a_name_non_void: a_name /= void
    non_empty_a_name: not a_name.is_empty
    not_already_owned: not is_room_owned (a_name)
  local
    a_room: CHAT_ROOM
  do
    create a_room.make (a_name, Current)
    owned.force (a_room)
    Result := a_room
  ensure
    room_create: Result.name.is_equal (a_name)
    ownership_added: owned_model |=| (old owned_model).extended_by (Result)
    is_allowed_in_room: Result.is_user_allowed (user_name)
  end

add_room (a_room: CHAT_ROOM) is
    -- add ’a_room’ to the chat server
  require
    a_room_non_void: a_room /= Void
    user_connected: is_connected
    user_is_owner: owned.has (a_room)
    room_is_new: not server.has_room (a_room.name)
  do
    server.add_room (a_room, Current)
  ensure
    room_added: server.ownership_model.domain.has (a_room)
    owned_unchaged: owned_model |=| old owned_model
  end

remove_room (a_room: CHAT_ROOM) is
    -- user removes a room
  require
    room_non_void: a_room /= void
    user_connected: is_connected
    user_is_owner: is_room_owned (a_room.name)
  do
    server.remove_room (Current, a_room.name)
  ensure
    not_has_it: not server.has_room (a_room.name)
  end

enter_room (a_room: STRING) is
    -- user enters room 'a_room'
  require
    a_room_non_void: a_room /= Void and not a_room.is_empty
    room_exists: server.has_room (a_room)
    user_allowed: server.is_allowed (Current, a_room)
    user_not_already_in: not room.name.is_equal (a_room)
  do
    server.enter_room (Current, a_room)
  ensure
    entered_room: room.name.is_equal (a_room)
    owned_unchaged: owned_model |=| old owned_model
  end

set_private (a_room: STRING) is
  require
    a_room_non_void: a_room /= Void and not a_room.is_empty
    user_is_owner: is_room_owned (a_room)
    room_not_private: not server.is_room_private (a_room)
  do
    server.set_private (Current, a_room)
  ensure
    is_private: server.is_room_private (a_room)
    owned_unchaged: owned_model |=| old owned_model
  end

set_public (a_room: STRING) is
  require
    a_room_non_void: a_room /= Void and not a_room.is_empty
    user_is_owner: is_room_owned (a_room)
    room_not_public: server.is_room_private (a_room)
  do
    server.set_public (Current, a_room)
  ensure
    is_public: not server.is_room_private (a_room)
    owned_unchaged: owned_model |=| old owned_model
  end
  allow_user (a_user: STRING; a_room: STRING) is
      -- allow 'a_user' to access 'a_room'
    require
      a_user_non_void: a_user /= Void and not a_user.is_empty
      a_room_non_void: a_room /= Void
      user_is_owner: is_room_owned (a_room)
    do
      server.allow_user (a_user, a_room, Current)
    ensure
      user_allowed: get_owned_room (a_room).is_user_allowed (a_user)
      owned_unchaged: owned_model |=| old owned_model
    end

feature {NONE} -- agents

  has_room_with_name (a_room: CHAT_ROOM; a_name: STRING): BOOLEAN is
      -- true if name of 'a_room' is equal to 'a_name'
    do
      Result := a_room.name.is_equal (a_name)
    end

  get_owned_room (a_room: STRING): CHAT_ROOM is
      -- returns the room associated with the a_room
    require
      a_room_non_void: a_room /= Void
      a_room_is_owned: is_room_owned (a_room)
    local
      loc: INTEGER
      found: BOOLEAN
    do
      from
        loc := owned.index
        owned.start
      until
        owned.after or found
      loop
        if owned.item.name.is_equal (a_room) then
          found := true
          Result := owned.item
        end
        owned.forth
      end
      owned.go_i_th (loc)
    end
  server_has_room (a_room: CHAT_ROOM): BOOLEAN is
      -- server does not have 'a_room'
    do
      Result := server.has_room (a_room.name)
    end

invariant
  user_name_non_void: user_name /= Void
  owned_non_void: owned /= Void
end
```

B.7 CHAT_SERVER

```
class CHAT_SERVER create
  make

feature {UNIT_TEST, ES_FIXTURE_UNIT} -- access
  lobby: CHAT_ROOM  -- Lobby room of the server
  admin: CHAT_USER  -- Administrator user
  rooms: LIST[CHAT_ROOM] -- list of Server chatrooms
  users: LIST[CHAT_USER] -- list of Server users
  Admin_name: STRING is "Admin"
  Lobby_name: STRING is "Lobby"

feature {UNIT_TEST, ES_FIXTURE_UNIT} -- creation
  make is
    do
      rooms := create {LINKED_LIST[CHAT_ROOM]}.make
      users := create {LINKED_LIST[CHAT_USER]}.make
      create admin.make (admin_name)  -- create admin user
      users.force (admin)  -- add it to the server
      admin.set_server (Current)
                     -- admin creates lobby
      lobby := admin.create_room (lobby_name)
      lobby.set_server (Current)
      rooms.force (lobby)    -- admin enters lobby
      lobby.occupants.force (admin)
      admin.set_room (lobby)
    ensure
      admin_in_lobby: location_model |=| (old location_model).extended_by (admin,
           lobby)
      admin_owns_lobby: ownership_model |=| (old ownership_model).extended_by (
          lobby, admin)
      room_server_set: lobby.server = Current
      admin_server_set: admin.server = Current
      admin_name_set: admin.user_name.is_equal (Admin_name)
      lobby_name_set: lobby.name.is_equal (Lobby_name)
    end

feature -- queries

  is_active: BOOLEAN is
      -- returns true if server is active
    do
      Result := rooms.count >= 1
    ensure
      result_correct: # location_model >= 1
    end
```

```
user_count: INTEGER is
    -- number of users on the chat server
  do
    Result := users.count
  end

room_count: INTEGER is
    -- number of rooms on the chat server
  do
    Result := rooms.count
  end

has_user (a_name: STRING): BOOLEAN is
    -- returns true if chat server has a user with username 'a_name'
  require
    a_name_non_void: a_name /= Void
  local
    loc: INTEGER
  do
    from
      loc := users.index -- for consistancy keep the current index
      users.start
    until
      users.after or Result
    loop
      if users.item.user_name.is_equal (a_name) then
        Result := true
      end
      users.forth
    end
    users.go_i_th (loc) -- set the index to original
  ensure
    result_correct: Result = location_model.domain.there_exists (agent
        user_with_name (?, a_name))
  end

has_room (a_name: STRING): BOOLEAN is
    -- returns true if chat server has a room with name 'a_name'
  require
    a_name_non_void: a_name /= void
  local
    loc: INTEGER
  do
    from
      loc := rooms.index
      rooms.start
      until
        rooms.after or Result
      loop
        if rooms.item.name.is_equal (a_name) then
          Result := true
        end
        rooms.forth
      end
      rooms.go_i_th (loc)
    ensure
      result_correct: Result = ownership_model.domain.there_exists (agent
          room_with_name (?, a_name))
    end

  is_allowed (a_user: CHAT_USER; a_room: STRING): BOOLEAN is
      -- is 'a_user' allowed to access 'a_room'?
    require
      a_room_non_void: a_room /= void and not a_room.is_empty
      user_exists: location_model.has_key (a_user)
      room_exists: has_room (a_room)
    local
      the_room: CHAT_ROOM
    do
      the_room := get_room (a_room)
      Result := the_room.is_allowed (a_user)
    ensure
      result_allowed: get_room (a_room).is_allowed (a_user)
    end

  is_room_private (a_room: STRING): BOOLEAN is
      -- returns true if room is private
    require
      a_room_non_void: a_room /= void and not a_room.is_empty
      room_exists: has_room (a_room)
    local
      the_room: CHAT_ROOM
    do
      the_room := get_room (a_room)
      Result := the_room.is_private
    end

feature {UNIT_TEST, ES_FIXTURE_UNIT} -- queries (private)

  get_user (a_name: STRING): CHAT_USER is
      -- returns a user with user_name 'a_name'
    require
      name_non_void: a_name /= Void
  already_in: has_user (a_name)
 local
  found: BOOLEAN
  loc: INTEGER
 do
  from
   loc := users.index
   users.start
  until
   users.after or found
  loop
   if (users.item).user_name.is_equal (a_name) then
    found := true
    Result := users.item
   end
   users.forth
  end
  users.go_i_th (loc)
 ensure
  server_has_it: location_model.has_key (Result)
  result_correct: Result.user_name.is_equal (a_name)
  locations_un_changed: location_model |=| old location_model
  ownership_un_changed: ownership_model |=| old ownership_model
 end

get_room (room_name: STRING): CHAT_ROOM is
  -- returns a room with name 'a_name'
 require
  name_non_void_non_empty: room_name /= Void
  already_in: has_room (room_name)
 local
  espec_loop: ESR_COMP[CHAT_ROOM]
  loc: INTEGER
 do
  loc := rooms.index
  create espec_loop
  Result := (espec_loop.for_list (rooms, agent name_equal(?, ?, room_name)))
    [1]
  imp_comment("Result = (those i: 1 .. rooms.count yield rooms[i] ~ room_name
    )")
  rooms.go_i_th (loc)
 ensure
  result_correct: ownership_model.there_exists (agent room_exists (?, Result,
     room_name))
  Result.name.is_equal (room_name) and ownership_model.has_key (Result)
  comment ("Result.name ~ room_name and Result in ownership_model.domain")
  locations_un_changed: location_model |=| old location_model
```

```
      ownership_un_changed: ownership_model |=| old ownership_model
    end

  get_room2 (a_name: STRING): CHAT_ROOM is
      -- returns a room with name 'a_name'
    require
      name_non_void_non_empty: a_name /= void and not a_name.is_empty
      already_in: has_room (a_name)
    local
      found: BOOLEAN
    do
      from rooms.start
      invariant
        found implies mSlice (rooms.index).there_exists (agent room_exists (?,
            Result, a_name))
        not found implies Result = Void
      variant
        # ownership_model - rooms.index + 1
      until rooms.after or found
      loop
        if (rooms.item).name.is_equal (a_name) then
          found := true
          Result := rooms.item
        end
        rooms.forth
      end
    ensure
      result_correct: ownership_model.there_exists (agent room_exists (?, Result,
           a_name))
    end

feature {NONE} -- Agents

  user_with_name (a_user: CHAT_USER; a_name: STRING): BOOLEAN is
      -- true if 'a_user's name is 'a_name'
    require
      a_user_non_void: a_user /= void
      a_name_non_void: a_name /= void
    do
      Result := a_user.user_name.is_equal (a_name)
    ensure
      Result = a_user.user_name.is_equal (a_name)
    end

  room_with_name (a_room: CHAT_ROOM; a_name: STRING_8): BOOLEAN is
      -- true if 'a_room's name is 'a_name'
    require
```

```
      a_room_non_void: a_room /= void
      a_name_non_void: a_name /= void
    do
      Result := a_room.name.is_equal (a_name)
    ensure
      Result = a_room.name.is_equal (a_name)
    end

  name_equal (r: CHAT_ROOM; i: INTEGER; name: STRING): BOOLEAN is
    do
      Result := r.name.is_equal (name)
--   Result := rooms[i].name.is_equal(name)
    end

  mSlice (upto: INTEGER): ML_MAP [CHAT_ROOM, CHAT_USER] is
      -- creates a slice of ownership map up to index 'upto'
    local
      seq_room_users: ML_SEQ [ML_PAIR[CHAT_ROOM, CHAT_USER]]
      i: INTEGER
    do
      from
        create Result.make
        seq_room_users := ownership_model.to_seq
        i := 1
      until
        i = upto
      loop
        Result := Result.extended_by_pair (seq_room_users.item (i))
        i := i + 1
      end
    end

  room_exists (r1: CHAT_ROOM; r2: CHAT_ROOM; name: STRING): BOOLEAN is
    do
      Result := r1 = r2 and r1.name.is_equal (name)
    end

  user_name_is_unique (a_user: CHAT_USER): BOOLEAN is
      -- username of 'a_user' is unique
    require
      a_user_non_void: a_user /= Void
    do
      Result := not location_model.there_exists (agent same_name (?, a_user))
    end

  same_name (user1: CHAT_USER; user2: CHAT_USER): BOOLEAN is
      -- true if user1 /= user2 but user1.user_name.is_equal (user2.user_name)
    do
      Result := user1 /= user2 and user1.user_name.is_equal (user2.user_name)
    end

  room_name_is_unique (a_room: CHAT_ROOM): BOOLEAN is
      -- room name of 'a_room' is unique
    require
      a_room_non_void: a_room /= Void
    local
      loc: INTEGER
    do
      from
        loc := rooms.index
        Result := true
        rooms.start
      until
        rooms.after or not Result
      loop
        if rooms.item /= a_room and rooms.item.name.is_equal (a_room.name) then
          Result := false
        end
        rooms.forth
      end
      rooms.go_i_th (loc)
    end


  comment(s:STRING):BOOLEAN do Result := true end
  imp_comment(s:STRING) do end

feature -- models

  location_model: ML_MAP [CHAT_USER, CHAT_ROOM] is
      -- model for the chat server
    local
      loc:INTEGER
    do
      from
        create Result.make
        if users /= void then
          -- store cursor's location
          loc := users.index
          users.start
         end
      until
        users = void or else users.after
      loop
        Result := Result.extended_by (users.item, users.item.room)
        users.forth
      end
      if users /= Void then -- set the cursor to its original position
        users.go_i_th (loc)
      end
    end

  ownership_model: ML_MAP [CHAT_ROOM, CHAT_USER] is
      -- maps the rooms to the owners
    local
      loc: INTEGER
    do
      from
        create Result.make
        if rooms /= void then
          loc := rooms.index
          rooms.start
        end
      until
        rooms = void or else rooms.after
      loop
        Result := Result.extended_by (rooms.item, rooms.item.owner)
        rooms.forth
      end

      if rooms /= Void then -- set the cursor to its original position
        rooms.go_i_th (loc)
      end
    end

  occupants (r: CHAT_ROOM): ML_SET[CHAT_USER] is
      -- returns a model of ’r’s occupants
    local
      loc: INTEGER
    do
      from
        create Result.make
        loc := r.occupants.index
        r.occupants.start
      until
        r.occupants.after
      loop
        Result := Result.extended_by (r.occupants.item)
        r.occupants.forth
      end
      r.occupants.go_i_th (loc)
    end

feature {CHAT_USER, UNIT_TEST, ES_FIXTURE_UNIT} -- Server Actions

  connect (a_user: CHAT_USER) is
      -- connect ’a_user’ to chat server if user not already connected
    require
      user_non_void: a_user /= Void and a_user.user_name /= Void
      user_not_connected: not a_user.is_connected
      user_name_not_in: not location_model.has_key (a_user)
    do
      users.force (a_user)
      a_user.set_server (Current)
      lobby.occupants.force (a_user)
      a_user.set_room (lobby)
    ensure
      user_added: location_model |=| (old location_model).extended_by (a_user,
          lobby)
      ownership_un_changed: ownership_model |=| old ownership_model
      user_connected: a_user.server = Current
      user_in_lobby: a_user.room = lobby
      lobby_has_user: lobby.occupants.has (a_user)
    end

  add_room (a_room: CHAT_ROOM; a_user: CHAT_USER) is
      -- adds the ’a_room’ to the current server
    require
      a_room_non_void: a_room /= Void and a_room.name /= Void
      a_user_non_void: a_user /= Void and a_user.user_name /= Void
      requester_is_owner: a_user.is_room_owned (a_room.name)
      has_user: location_model.has_key (a_user)
      not_already_in: not ownership_model.has_key (a_room)
    do
      rooms.force (a_room)
      a_room.set_server (Current)
    ensure
      current_server_set: a_room.server = Current
      ownership_model_updated: ownership_model |=| (old ownership_model).
          extended_by (a_room, a_user)
      locations_un_changed: location_model |=| old location_model
    end

  remove_room (a_user: CHAT_USER; a_room: STRING) is
      -- remove a room from the server
    require
      user_non_void: a_user /= void
      a_room_non_void: a_room /= void
      room_exists: has_room (a_room)
      has_user: location_model.has_key (a_user)
      user_is_owner: is_allowed (a_user, a_room)
    do
      help_remove_room (a_user, get_room (a_room))
    end


  enter_room (a_user: CHAT_USER; a_room: STRING) is
      -- ’a_user’ requests to enter into ’a_room’
    require
      a_user_non_void: a_user /= void
      a_room_non_void: a_room /= void and not a_room.is_empty
      room_exists: has_room (a_room)
      has_user: location_model.has_key (a_user)
      not_already_in: not a_user.is_in_room (a_room)
      user_allowed: is_allowed (a_user, a_room)
    local
      the_room: CHAT_ROOM
--    dummy_user: CHAT_USER
    do
      the_room := get_room (a_room)
      a_user.room.remove_user (a_user)
      the_room.add_user (a_user)
--    create dummy_user.make ("Dummy")
--    the_room.add_user (dummy_user)
    ensure
      user_entered: location_model |=| (old location_model).override (a_user,
          get_room (a_room))
      ownerships_not_changed: ownership_model |=| old ownership_model
    end

  set_private (a_user: CHAT_USER; a_room: STRING) is
      -- ’a_user’ sets ’a_room’ to private, this removes all unauthorized users
    require
      user_non_void: a_user /= void
      a_room_non_void: a_room /= void and not a_room.is_empty
      room_exists: has_room (a_room)
      has_user: location_model.has_key (a_user)
      user_is_owner: a_user.is_room_owned (a_room)
      room_public: not is_room_private (a_room)
    local
      the_room: CHAT_ROOM
      temp_user: CHAT_USER
      loc: INTEGER
    do
      the_room := get_room (a_room)
      -- move all un-authorized users to the lobby
      from
        loc := the_room.occupants.index
        the_room.occupants.start
      until
        the_room.occupants.after
      loop
        temp_user := the_room.occupants.item
        if not the_room.is_in_allowed_list (temp_user) then
          the_room.remove_user (temp_user)
          lobby.add_user (temp_user)
        else
          the_room.occupants.forth
        end
      end
      the_room.set_private
      the_room.occupants.go_i_th (loc)
    ensure
      room_set_to_private: get_room (a_room).is_private
      all_unauthorized_move_to_lobby: (old location_model).comp (agent
          in_room_not_allowed (?, ?, a_room)).for_all (agent user_room_is_lobby
          (?, ?))
      authorized_in_room: (old location_model).comp (agent in_room_allowed (?, ?,
           a_room)) |=| (location_model).comp (agent in_room_allowed (?, ?,
          a_room))
      others_unchanged: (old location_model).comp (agent not_in_room (?, ?,
          a_room)) |=| (location_model.comp (agent not_in_room (?, ?, a_room))).
          difference ((old location_model).comp (agent in_room_not_allowed (?, ?,
           a_room) ) )
      owner_model_unchanged: ownership_model |=| old ownership_model
    end

  set_public (a_user: CHAT_USER; a_room: STRING) is
      -- sets the room to public
    require
      user_non_void: a_user /= void
      a_room_non_void: a_room /= void and not a_room.is_empty
      room_exists: has_room (a_room)
      has_user: location_model.has_key (a_user)
      user_is_owner: a_user.is_room_owned (a_room)
      room_private: is_room_private (a_room)
    local
      the_room: CHAT_ROOM
    do
      the_room := get_room (a_room)
      the_room.set_public
    ensure
      room_set_to_public: not get_room (a_room).is_private
    end

  allow_user (a_user: STRING; a_room: STRING; requester: CHAT_USER) is
      -- a 'requester' requests the server to allow 'a_user' to access 'a_room'
    require
      a_user_non_void: a_user /= void
      a_room_non_void: a_room /= void and not a_room.is_empty
      room_exists: has_room (a_room)
      has_user: has_user (a_user)
      user_is_owner: requester.is_room_owned (a_room)
    local
      the_room: CHAT_ROOM
      temp_user: CHAT_USER
    do
      the_room := get_room (a_room)
      temp_user:= get_user (a_user)
      the_room.allow_user (temp_user)
    ensure
      user_allowed: is_allowed (get_user(a_user), a_room)
    end

  disconnect (a_user: CHAT_USER) is
      -- disconnects the user to chat server
      -- (a) it should remove all rooms belonging to this user
      -- (b) move all users from such rooms to lobby
    require
      user_non_void: a_user /= void
      user_in: location_model.has_key (a_user)
      user_connected: a_user.is_connected
    do
      -- close all rooms whose owner is 'a_user'
      remove_all_rooms_of (a_user)  -- close all rooms with owner 'a_user'
      a_user.room.remove_user (a_user)-- remove the user from its current room
      users.start      -- remove the user from the server user's list
      users.search (a_user)
      users.remove
      users.start
    ensure
      user_disconnected: not a_user.is_connected
      no_room_is_owned_by_this_user: not ownership_model.domain.there_exists (
          agent room_owned_by (?, a_user))
      all_users_in_owned_rooms_moved_to_lobby: true
    end

  remove_all_rooms_of (a_user: CHAT_USER) is
```

```
    -- closes all rooms whose owner is 'a_user'
  require
    a_user_non_void: a_user /= Void
    has_user: location_model.has_key (a_user)
  do
    from
      rooms.start
    until
      rooms.after
    loop
      if rooms.item.owner = a_user then
        help_remove_room (a_user, rooms.item)
      else
        rooms.forth
      end
    end
  ensure
    rooms_removed: not ownership_model.domain.there_exists (agent room_owned_by
          (?, a_user))
    all_rooms_removed: ownership_model.union (old ownership_model.comp (agent
        pairs_affected (?, ?, a_user))) |=| old ownership_model
  end

list_room_names (a_user: CHAT_USER): LINKED_LIST[STRING] is
    -- a user may list room names of a chat server (lists only show public
        rooms)
    -- private rooms are shown to allowed list
  require
    a_user_non_void: a_user /= void
    has_user: location_model.has_key (a_user)
  local
    loc: INTEGER
  do
    from
      loc := rooms.index
      rooms.start
      create Result.make
    until
      rooms.after
    loop
      if (not rooms.item.is_private) or (rooms.item.is_private and rooms.item.
          is_allowed (a_user)) then
        Result.force (rooms.item.name)
      end
      rooms.forth
    end
    rooms.go_i_th (loc)
    end


feature {NONE} -- agents

  in_room_not_allowed (a_user: CHAT_USER; a_room: CHAT_ROOM; name: STRING):
      BOOLEAN is
    do
      Result := a_room.name.is_equal (name) and not a_room.is_allowed (a_user)
    end

  user_room_is_lobby (a_user: CHAT_USER; a_room: CHAT_ROOM): BOOLEAN is
    do
      Result := a_user.room = lobby
    end

  not_in_room (a_user: CHAT_USER; a_room: CHAT_ROOM; name: STRING): BOOLEAN is
    do
      Result := not (a_room.name.is_equal (name))
    end

  in_room_allowed (a_user: CHAT_USER; a_room: CHAT_ROOM; name: STRING): BOOLEAN
      is
    do
      Result := a_room.name.is_equal (name) and a_room.is_allowed (a_user)
    end

  room_owned_by (a_room: CHAT_ROOM; a_user: CHAT_USER): BOOLEAN is
      -- returns true if room is owned by user
    do
      Result := a_room.owner = a_user
    end

  collect_people_in_room (a_user: CHAT_USER; a_room: CHAT_ROOM; other_room:
      CHAT_ROOM): BOOLEAN is
      -- returns true if 'a_room' = 'other_room'
    require
      a_user_non_void: a_user /= void
      a_room_non_void: a_room /= void
      other_room_non_void: other_room /= void
    do
      Result := (a_room = other_room)
    ensure
      Result = (a_room = other_room)
    end

  room_associated_user_is_lobby (a_user: CHAT_USER): BOOLEAN is
```

```
      -- returns true if 'a_user's current room is lobby
    require
      a_user_non_void: a_user /= void
    do
      Result := a_user.room = lobby
    ensure
      Result = (a_user.room = lobby)
    end

  mapping_changed_or_maintained (user: CHAT_USER; room: CHAT_ROOM; a_room:
      CHAT_ROOM;): BOOLEAN is
      -- returns true if desired room's location in lobby
    do
      if room = a_room then
        Result := (location_model[user] = lobby)
      else
        Result := (location_model[user] = room)
      end
    end

  pairs_affected (a_room: CHAT_ROOM; a_user: CHAT_USER; owner: CHAT_USER):
      BOOLEAN is
      -- if a_user = owner
    do
      Result := a_user = owner
    end


feature {NONE} -- agents for the loop

  sublist_of_rooms (upto: INTEGER): LINKED_LIST[CHAT_ROOM] is
      --
    local
      i: INTEGER
      pos: INTEGER
    do
      from
        i := 1
        pos := rooms.index
        create Result.make
        rooms.start
      until
        i = upto
      loop
        Result.extend (rooms.item)
        rooms.forth
        i := i + 1
      end

      rooms.go_i_th (pos)
    end

  sublist_of_users (upto: INTEGER): LINKED_LIST[CHAT_USER] is
      --
    local
      i: INTEGER
      pos: INTEGER
    do
      from
        i := 1
        pos := users.index
        create Result.make
        users.start
      until
        i = upto
      loop
        Result.extend (users.item)
        users.forth
        i := i + 1
      end

      users.go_i_th (pos)
    end

  room_with_name_exists (a_room: CHAT_ROOM; a_user: CHAT_USER; room_name: STRING
      ): BOOLEAN is
      --
    do
      Result := a_room.name.is_equal (room_name)
    end


    forall_rooms (r1: CHAT_ROOM): BOOLEAN is
        do
            Result := ownership_model.domain.for_all (agent empty_intersection (r1
                , ?))
        end

    empty_intersection (r1, r2: CHAT_ROOM): BOOLEAN is
        do
            if r1 /= r2 then
                Result := (r1.occupant_model * r2.occupant_model) |=| create {
                    ML_SET[CHAT_USER]}.make
            else
```

```
                Result := true
            end
        end

    multi_union (s: ML_SET[CHAT_ROOM]): ML_SET[CHAT_USER] is
      local
        seq: ML_SEQ[CHAT_ROOM]
      do
        seq := s.to_seq
        if seq.count = 1 then
          Result := seq.head.occupant_model.to_set
        else
          Result := (multi_union (seq.tail.to_set) |++ (seq.head.occupant_model))
              .to_set
        end
      end

feature {NONE} -- helpers

  help_remove_room (a_user: CHAT_USER; a_room: CHAT_ROOM) is
      -- removes 'a_room' requested by 'a_user'

    do
      move_users_from_room_to_lobby (a_room)
      rooms.start
      rooms.search (a_room)
      rooms.remove
    ensure
      room_removed: ownership_model.extended_by (a_room, a_user) |=| old
          ownership_model
--   no_user_owns_the_removed_room: not user_to_room_model.range_bag.has (a_room
    )
--   The above contract is violated since when removing a room owner remains
    unchanged: comment first line out
--   label: (old user_to_room_model - user_to_room_model).domain |=| old
    user_to_room_model.comp (agent collect_people_in_room (?, ?, a_room)).domain
      all_moved_to_lobby: (old location_model.comp (agent collect_people_in_room
          (?, ?, a_room)).domain).for_all (agent room_associated_user_is_lobby
          (?))
      other_remained_the_same: (old location_model).for_all (agent
          mapping_changed_or_maintained (?, ?, a_room))
    end

  move_users_from_room_to_lobby (a_room: CHAT_ROOM) is
      -- move all users of the current room to the
      -- lobby room
    require
```

```
      a_room_non_void: a_room /= void
      room_exists: ownership_model.has_key (a_room)
    local
      loc: INTEGER
    do
      from
        loc := users.index
        users.start
      until
        users.after
      loop
        if users.item.room = a_room then
          enter_room (users.item, lobby_name)
        end
        users.forth
      end
      users.go_i_th (loc)
    ensure
      ownership_not_changed: ownership_model |=| old ownership_model
      all_moved_to_lobby: (old location_model.comp (agent collect_people_in_room
          (?, ?, a_room)).domain).for_all (agent room_associated_user_is_lobby
          (?))
    end

  switch_user_to_lobby (a_user: CHAT_USER) is
      -- moves 'a_user' from its current room to looby
    require
      a_user_non_void: a_user /= void
      user_not_already_in_lobby: not lobby.has_user (a_user)
    do
      enter_room (a_user, "Lobby")
    ensure
      user_in_lobby: lobby.has_user (a_user)
    end


invariant
  pairwise_disjoint: ownership_model.domain.for_all (agent forall_rooms (?))
  coverage: multi_union (ownership_model.domain) |=| location_model.domain
  user_names_unique: location_model.domain.for_all (agent user_name_is_unique
      (?))
  room_names_unique: ownership_model.domain.for_all (agent room_name_is_unique
      (?))
  administrator_non_void: admin /= Void
  lobby_non_void: lobby /= Void

end
```

C Appendix: ES-Verify

C.1 Introduction

A software product is reliable if it is correct (performs its tasks according to specification) and robust (reacts appropriately to abnormal conditions). How should specifications be provided and how do we check that software behaves according to its specification? Design by Contract (DbC) is a promising method for answering these questions. A class can be specified via expressive pre-conditions, post-conditions and class invariants [67].

A variety of object-oriented languages have followed this contracting approach to software quality such as Eiffel [67], Spec# [8], JML [60] tools like ESC/-Java2 [28, 21], and UML/OCL [17]. A “lightweight” formal approach to checking the correctness of code works by runtime assertion checking, i.e. the contracts are checked as the code is executed and an exception is raised if there is a contract violation. However, we would also like to reason formally about the correctness of programs and to mechanize such process. Automated verification of object-oriented code has been pursued in systems such as Spec# and JML tools like ESC/Java2.

ESpec (Eiffel Specification) software quality workbench is a unified environment allowing software developers to combine Fit tables (ES-Fit for customer requirements and acceptance tests) with contracts and unit testing tools (ES-Test). This means that a single integrated tool can be used to specify, develop, test, and verify the requirements and design of a software product. Formal verification is a substantial addition to the capabilities of the ESpec toolset, allowing for a combination of lightweight validation and automated deductive verification.

In this chapter we describe the automated model-based verification for a significant subset of Eiffel. The following three components, which together we call the ES-Verify, are under development as part of the ESpec suite:

- An Eiffel Model Library (ML) for specifying the abstract state of a program without exposing its implementation details. This library is similar to the model-based specifications as in B [1] and Z [83], except that it is object-oriented. ML contains classes such as `ML_SEQ`, `ML_SET`, `ML_BAG`, and `ML_MAP`. These classes are both immutable and executable. They are immutable so that software properties specified in the pre- and post- conditions as well as the class invariants can be based on them. They are executable so that contract violations will be reported (if any). This mathematical library is thus useful for lightweight verification even in the absence of a theorem prover.

- An Eiffel base library (`ES_BASE`) of data structures (classes such as `ESV_ARRAY`, `ESV_LIST`, `ESV_SET`, and `ESV_TABLE`) for the efficient implementation of software products. The prefix "ESV" stands for an "ESpec Value" structure, which is part of the ESpec library (built on top of the Eiffel base library via inheritance) for implementing code. These ESV classes apply a value semantics [30], but for efficiency they are mutable. While class features are contracted via ML (which are executable but inefficient due to their mathematical immutability), their bodies are implemented via the `ES_BASE` classes (which are mutable and hence efficient, but not as suitable for specifications as ML ones).

- A translator that will convert Eiffel code implemented via `ES_BASE` and specified via ML into an equivalence written in a specification language Perfect [35]. The advantage of this translator is that there is, associated with the Perfect language, a fully-automated reasoning tool - Perfect Developer (PD) - that fits well for our source Eiffel code. PD supports object-oriented, model-driven, and DbC software development as well as its verification [29]. PD converts its specification (written in the Perfect Language) into complete verification conditions and attempts to automatically discharge their proofs.

As stated, ES-Verify uses the PD tools (the Perfect language and its associated theorem prover). Although we are impressed by the expressiveness and power of the PD tools, we have not used them in the intended fashion. The intended use of PD tools is that developers write their specifications in the Perfect Language, which is then used to automatically generate executable code (e.g. Java or C++). In this respect, Perfect is akin to model-driven development (MDD) methods. Perfect also has a notion of refinement that can be used to improve the efficiency of the generated code.

We have examined the Java code and found that the generated code - much longer and more complex than the original contract-based specification - is not intended to be read. The MDD approach is useful if there is never a need to deal with the generated code. However, Perfect specifications are neither directly executable nor is there a debugger at the model level. As a result, our preference is to write code in Eiffel. Eiffel has a mature industrial-strength contracting mechanism with a full set of tools such as debuggers, profilers, documentation, and browsing capabilities. The language is admired for its clear syntax and expressive use of a full range of object-oriented constructs such as multiple inheritance.

Our approach is to write the code in Eiffel and thus retaining the simple but expressive use of its language constructs. The Eiffel code is then translated into Perfect using (a) the Perfect refinement constructs for Eiffel feature implementations and (b) the Perfect contracting mechanism for Eiffel contracts. The Eiffel model library (ML) was designed in order to avoid mismatches between itself and the Perfect data structures. Theorem proving program involving genericity and loops (with their invariants) is a non-trivial task, and this work shows that model libraries (such as ML) must be designed with the target theorem prover in mind. In the sequel we will use the abbreviation PD for the combination the Perfect specification language and its associated theorem prover.

C.2 Models via ML

As explained in [83] with reference to Z, formal specifications use mathematical notation to describe, in a precise way, the properties which a software product

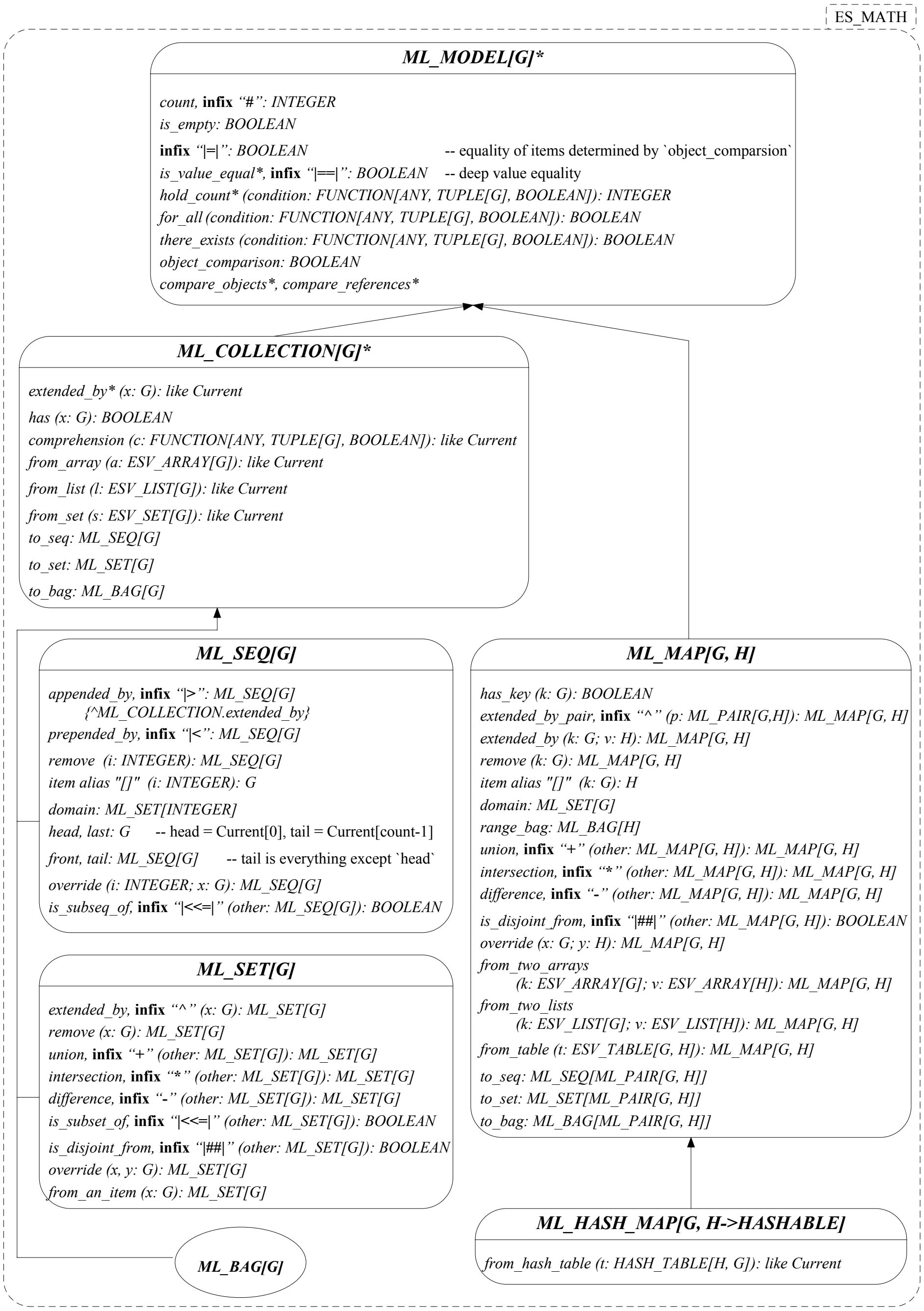
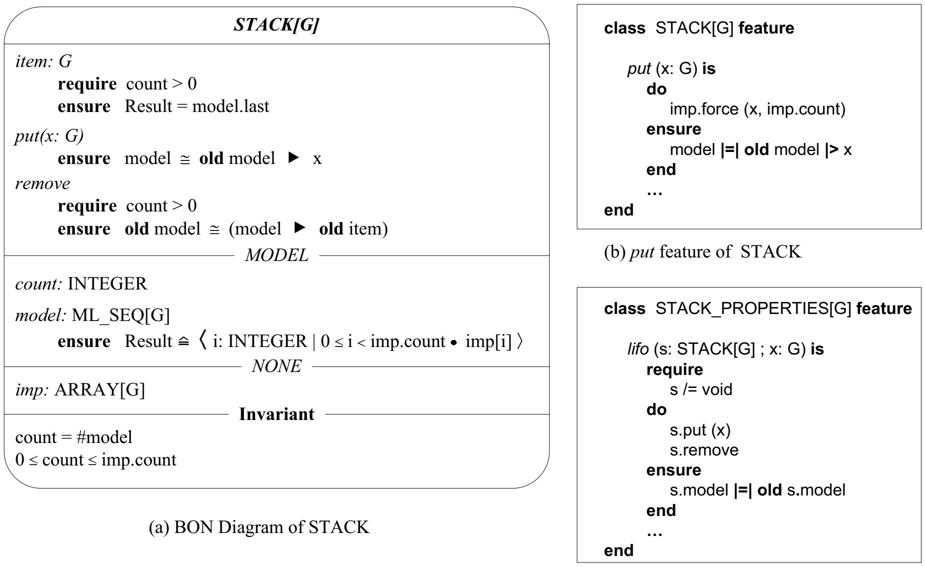

ES_MATH
ML_MODEL[G]*
count, infix "#": INTEGER
is_empty: BOOLEAN
infix "|=|": BOOLEAN -- equality of items determined by `object_comparsion`
is_value_equal*, infix "|==|": BOOLEAN -- deep value equality
hold_count* (condition: FUNCTION[ANY, TUPLE[G], BOOLEAN]): INTEGER
for_all (condition: FUNCTION[ANY, TUPLE[G], BOOLEAN]): BOOLEAN
there_exists (condition: FUNCTION[ANY, TUPLE[G], BOOLEAN]): BOOLEAN
object_comparison: BOOLEAN
compare_objects*, compare_references*
ML_COLLECTION[G]*
extended_by* (x: G): like Current
has (x: G): BOOLEAN
comprehension (c: FUNCTION[ANY, TUPLE[G], BOOLEAN]): like Current
from_array (a: ESV_ARRAY[G]): like Current
from_list (l: ESV_LIST[G]): like Current
from_set (s: ESV_SET[G]): like Current
to_seq: ML_SEQ[G]
to_set: ML_SET[G]
to_bag: ML_BAG[G]
ML_SEQ[G]
appended_by, infix "|>": ML_SEQ[G]
{^ML_COLLECTION.extended_by}
prepended_by, infix "|<": ML_SEQ[G]
remove (i: INTEGER): ML_SEQ[G]
item alias "[]" (i: INTEGER): G
domain: ML_SET[INTEGER]
head, last: G -- head = Current[0], tail = Current[count-1]
front, tail: ML_SEQ[G] -- tail is everything except `head`
override (i: INTEGER; x: G): ML_SEQ[G]
is_subseq_of, infix "|<<=|" (other: ML_SEQ[G]): BOOLEAN
ML_MAP[G, H]
has_key (k: G): BOOLEAN
extended_by_pair, infix "^" (p: ML_PAIR[G,H]): ML_MAP[G, H]
extended_by (k: G; v: H): ML_MAP[G, H]
remove (k: G): ML_MAP[G, H]
item alias "[]" (k: G): H
domain: ML_SET[G]
range_bag: ML_BAG[H]
union, infix "+" (other: ML_MAP[G, H]): ML_MAP[G, H]
intersection, infix "*" (other: ML_MAP[G, H]): ML_MAP[G, H]
difference, infix "-" (other: ML_MAP[G, H]): ML_MAP[G, H]
is_disjoint_from, infix "|##|" (other: ML_MAP[G, H]): BOOLEAN
override (x: G; y: H): ML_MAP[G, H]
from_two_arrays
(k: ESV_ARRAY[G]; v: ESV_ARRAY[H]): ML_MAP[G, H]
from_two_lists
(k: ESV_LIST[G]; v: ESV_LIST[H]): ML_MAP[G, H]
from_table (t: ESV_TABLE[G, H]): ML_MAP[G, H]
to_seq: ML_SEQ[ML_PAIR[G, H]]
to_set: ML_SET[ML_PAIR[G, H]]
to_bag: ML_BAG[ML_PAIR[G, H]]
ML_SET[G]
extended_by, infix "^" (x: G): ML_SET[G]
remove (x: G): ML_SET[G]
union, infix "+" (other: ML_SET[G]): ML_SET[G]
intersection, infix "*" (other: ML_SET[G]): ML_SET[G]
difference, infix "-" (other: ML_SET[G]): ML_SET[G]
is_subset_of, infix "|<<=|" (other: ML_SET[G]): BOOLEAN
is_disjoint_from, infix "|##|" (other: ML_SET[G]): BOOLEAN
override (x, y: G): ML_SET[G]
from_an_item (x: G): ML_SET[G]
ML_HASH_MAP[G, H->HASHABLE]
from_hash_table (t: HASH_TABLE[H, G]): like Current
ML_BAG[G]

Figure C.1: Classes in the Mathematical Library (ML)



(a) BON Diagram of STACK

```
class STACK[G] feature

    put (x: G) is
      do
         imp.force (x, imp.count)
      ensure
         model |=| old model |> x
      end
      ...
end
```

(b) *put* feature of STACK

```
class STACK_PROPERTIES[G] feature

    lifo (s: STACK[G] ; x: G) is
      require
         s /= void
      do
         s.put (x)
         s.remove
      ensure
         s.model |=| old s.model
      end
      ...
end
```

(c) Stack LIFO property

Figure C.2: `STACK[G]` modelled by `ML_SEQ[G]`

must have, without unduly constraining the way in which these properties are achieved. We may call the mathematical description an abstract *model* of the system under development. The model describes *what* the system must do without saying *how* it is to be done. Models allow questions about what the system does to be answered confidently, without the need to either disentangle the information from a mass of detailed program code, or speculate about the meaning of phrases in an imprecisely-worded prose description.

In Z, the mathematical models are based on predicate logic and the set theory, and thus obey a rich collection of mathematical laws which makes it possible to effectively reason about the way a specified system will behave. But these models are not oriented towards computer representation.

The model library (ML) described in this chapter encodes predicate logic acting on sets, sequences, bags, and maps (as in Z), but the mathematical theories are structured as classes (instantiated to immutable objects needed for mathematical specification) whose features (e.g. $\forall$, $\exists$, $\in$, set comprehension, etc.) are pure functions executable in the object-oriented style. The Eiffel agent mechanism for iteratively applying a supplied expression to a collection is much used.

The classes of ML are shown in Fig. C.1. Contracts may be specified using ML and these contracts are executable. When runtime assertion checking is turned on, contract violations (if any) are signalled via exceptions, thus indicating an inconsistency between the implementation and its specification. The complete specification of a system and its implementation can be provided in the same compilable and executable Eiffel text (e.g. see class `STACK[G]` in Fig. C.7). The immutable ML classes will be inefficient (due to its re-construction of a new ML object every time a feature such as `appended_by` is invoked), by comparison to the mutable classes in the Eiffel or ES base library (such as `ARRAY` and `LIST`). But this is acceptable as contract checking may be turned off in the final delivered code which will only use the efficient base library for implementation.

As a simple example, consider the BON [87] contract view of a generic stack as shown in Fig. C.2a. The model of the stack consists of a `ML_SEQ[G]` (i.e. a sequence of items of type `G`, where `G` is a generic parameter) and `count` (the number of items in the stack). The contracts of all the other features of the stack can be described in terms of the sequence and `count`. In the absence of a sequence to model the stack (i.e. with just the model attribute `count`), the best post-condition for the stack push operation `put` is:

$$count = \textbf{old}\ \ count + 1\ \ \textbf{and}\ \ item = x \tag{C.1}$$

However, such abstract specification is incomplete. For example, an implementor can satisfy the above specification yet change old values of the stack that are not at the top. Therefore, we need a *frame condition* that says the old part of the stack remains unchanged. By adding a sequence to the model we can now express the complete contract as:

$$\mathit{model} \;\cong\; \textbf{old}\,\mathit{model} \;\blacktriangleright\; x \tag{C.2}$$

where ▶ is the `appended_by` (pure) function of a mathematical sequence that returns a new sequence same as the old one, but with the argument item appended to the end. Since (C.2) $\implies$ (C.1), there is then no need to write (C.1) as it is entailed by the model post-condition. With the full model we can then provide the complete contracts for the pop operation `remove` and the query `item` that returns the top of the stack. The Eiffel notation follows the BON notation quite closely as shown in Fig. C.2b. For ▶, we may use either the `appended_by` function or alternatively the infix operator `|>` as shown in class `ML_SEQ` in Fig. C.1.

Model classes such as `ML_SEQ` hold items that may be stored either by reference or by value. Eiffel has the **expanded** construct for constructing a value semantics. We thus introduce the notion of model equality (infix operator `|=|`) which depends on what type of comparison is requested (see `ML_MODEL` in Fig. C.1). The default is that two model sequences (say $s1$ and $s2$) are compared for their stored items via reference equality (i.e. `s1 |=| s2` iff the two sequences have the same size and the items stored at each index both refer to the same object). A specifier may invoke feature `compare_objects` (see `ML_MODEL`), in which case the items stored at each index will be compared based on how the inherited feature is_equal (of the actual generic type G) is defined.

With our contracts complete, and even in the absence of implementation details, we may already begin to validate our specification based only on the model. For example, the last-in-first-out (LIFO) property of the stack can be specified as shown in Fig. C.2c. In the absence of implementation, we cannot execute or unit test the LIFO property. However, with the translator and theorem prover, the LIFO property will prove with a warning that the body of put and remove must be refined with an implementation.

We must now refine the specification to an efficient implementation. We choose mutable structures such as an array or linked list. We may use ARRAY from the Eiffel base library, or from the ES base library if a value semantics rather than a reference semantics is desired (i.e. by declaring imp:ESV_ARRAY[G]).

Next, we need to define the abstraction relation between the abstract space in which the abstract program is written (i.e. model) and the space of the concrete representation (i.e. imp). This can be accomplished by giving an abstraction function which maps the concrete variables into the abstract objects which they represent. We may do this as follows. The body of the query model (a ML_SEQ[G]) for the stack in Fig. C.2 could be a loop that iterates through the implementation array and returns an equivalent sequence with the same elements as the array. That is, we "lift" the mutable array into a mathematical immutable sequence. The abstraction function [49] is captured by the post-condition of query model as follows:

$$Result = \langle i : INTEGER | 0 \leq i < imp.count.imp[i] \rangle \tag{C.3}$$

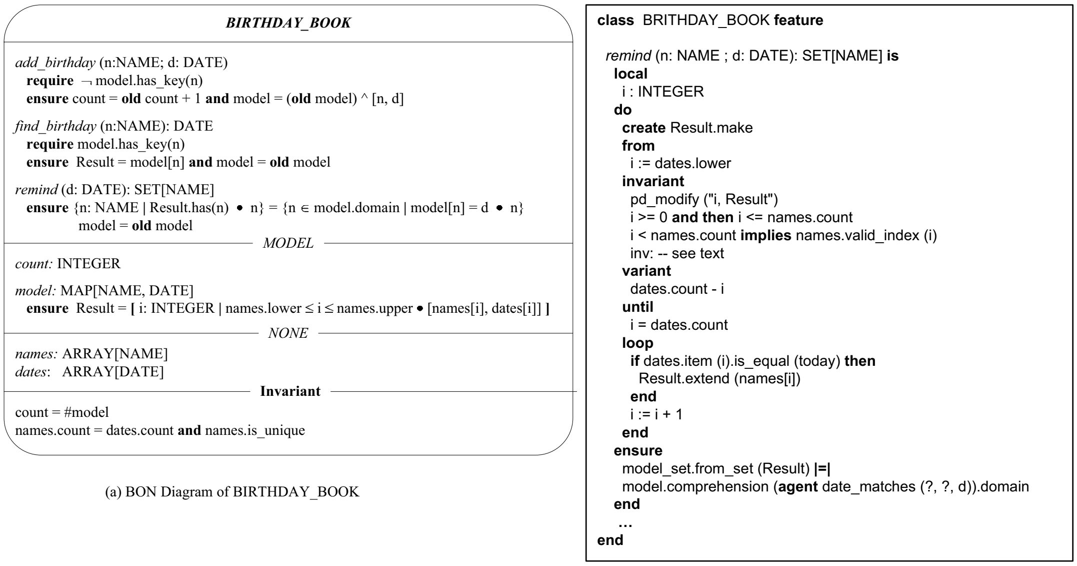
where the angle brackets $\langle\rangle$ stand for sequence comprehension in the same way that $\{\}$ stands for set comprehension. For example, $\{i : INT | 0 \leq i \leq 2i + 1\} = \{1, 2, 3\}$. Set, bag, sequence or map comprehension presents expressive notation for abstraction functions and is supported in ML. The Eiffel ML library uses the agent construct for writing comprehension (see Fig. C.1). However, for the post-condition of `model` we may use one of the pre-defined ML functions `from_array` that "lifts" an efficient mutable array to a mathematical sequence. Function `from_array` returns a new sequence whose items refer to the same items as in the array `imp` between $0 \cdots count - 1$. So the post-condition (C.3) written in ML becomes:

```
Result |=| Result.from_array(imp.subarray(0,count-1))
```

which asserts that the resulting sequence returned by the model is model-equal to the implementation array treated as a sequence. The contracts of all other features remain the same as they are all described in terms of `model`.

C.2.1 The Birthday Book example

The author of [85] reports that a web-enabled database system, consisting of 35,799 lines of Perfect, generated 9810 proof obligations and proved automatically in 4.5 hours (1.6 seconds per proof) on a modest laptop. We believe that the



```
BIRTHDAY_BOOK

add_birthday (n:NAME; d: DATE)
  require  ¬ model.has_key(n)
  ensure count = old count + 1 and model = (old model) ^ [n, d]

find_birthday (n:NAME): DATE
  require model.has_key(n)
  ensure  Result = model[n] and model = old model

remind (d: DATE): SET[NAME]
  ensure {n: NAME | Result.has(n)  •  n} = {n ∈ model.domain | model[n] = d  •  n}
             model = old model
────────────────────── MODEL ──────────────────────
count: INTEGER

model: MAP[NAME, DATE]
  ensure  Result = [ i: INTEGER | names.lower ≤ i ≤ names.upper • [names[i], dates[i]] ]
────────────────────── NONE ───────────────────────
names: ARRAY[NAME]
dates:   ARRAY[DATE]
──────────────────── Invariant ────────────────────
count = #model
names.count = dates.count and names.is_unique
```

(a) BON Diagram of BIRTHDAY_BOOK

```
class  BRITHDAY_BOOK feature

 remind (n: NAME ; d: DATE): SET[NAME] is
  local
   i : INTEGER
  do
   create Result.make
   from
    i := dates.lower
   invariant
    pd_modify ("i, Result")
    i >= 0 and then i <= names.count
    i < names.count implies names.valid_index (i)
    inv: -- see text
   variant
    dates.count - i
   until
    i = dates.count
   loop
    if dates.item (i).is_equal (today) then
     Result.extend (names[i])
    end
    i := i + 1
   end
  ensure
   model_set.from_set (Result) |=|
   model.comprehension (agent date_matches (?, ?, d)).domain
  end
   …
end
```

(b) *remind* feature of BIRTHDAY_BOOK

Figure C.3: Birthday Book

above performance is sustainable for reasonable chunks of code but there is minimal refinement and PD does the code generation. However, in our case there is refinement from high level models to more complex constructs (e.g. loops their variants and invariants), and thus the demands on PD are much greater. Nevertheless, by means of careful matching between ML and PD data structures as well as tuning of the translator, we can achieve proofs of the vast majority (if not all) verification conditions.

The birthday book example [83] nicely illustrates refinement to loops and more intensive use of ML as shown by the BON diagram in Fig. C.3a.

The model for the birthday book is a combination of the number of name-and-date pairs stored (i.e. `count`) and a `ML_MAP[NAME, DATE]` (i.e. a set of name-and-date pairs). Alternatively, this map is a function whose domain is a set of names and whose range is a bag of dates. The features of the birthday book include the ability to add a new pair (e.g. $[Peter, (March\ \ 1)]$), find a birthday given a name, and a `remind` function that for a given date d returns the set of names whose birthday is on d.

The `remind` function returns a set of names (`SET[NAME]`) where `SET` is an efficient mutable structure from either the Eiffel or ES base library. The birthday book is implemented as two arrays: one for names and the other for dates. The post-condition of the `remind` query is

$$\{n : NAME | Result.has(n) \bullet n\} = \{n \in model.domain | model[n] = d \bullet n\} \tag{C.4}$$

where the right hand side expression means the set of all names, from the domain of the model map, whose birthday is on the provided date d. And this must be equal to the left hand side expression which represents the set of all names returned by the `remind` function. The Eiffel notation for the `remind` function is shown in Fig. C.3b. The Eiffel post-condition of the `remind` query in (C.4) shown in Fig. C.4:

```
model_set.from_set(Result) |=| model.comprehension(agent date_matches (?, ?, d)
    ).domain}
```

Figure C.4: `remind` postcondition

The agent function used in the post-condition (and loop invariant) of the `remind` query is shown in Fig. C.5:

By defining a slice of the model map, according to the current loop counter i

```
date_matches (x: NAME; y, date: DATE): BOOLEAN is
    do
        if y.is_equal (date) then
            Result := true
        end
    end
```

Figure C.5: `remind` query

as well as arrays *names* and *dates*, as follows:

$$mSlice(i, names, dates) \triangleq \langle\langle j : INTEGER | 0 \leq j < i \bullet [names[j], dates[j]]\rangle\rangle \quad (C.5)$$

we can show that the loop invariant for the `remind` query has been constructed to approximate and hence similar to its post-condition:

$$\{n | Result.has(n) \bullet n\} = \{n \in mSlice(i, names, dates).domain | model[n] = d \bullet n\} \quad (C.6)$$

And the equivalent Eiffel loop invariant `inv` in Fig. C.3b) is shown in Fig. C.6.

```
model_set.from_set (Result) |=| model.from_two_arrays(names.subarray (0, i-1),
    dates.subarray(0, i-1)).comprehension(agent date_matches (?, ?, today)).
    domain
```

Figure C.6: Class invariant

C.3 The Eiffel to PD Translator

C.3.0.1 Underlying Theorem Prover

Our goal is to automatically verify Eiffel code specified via ML as in the stack and birthday book examples. The question would be, which theorem prover do we use? The *Perfect Developer* (PD) specification language and theorem prover [30] is a technically mature product that is aligned with the object-orientation and design by contract paradigms. PD theorem prover has about the same level of power and automation as *Simplify* [32] that is used for static verification in Spec# and ESC/Java2. *Simplify* handles integers and booleans at the primitive level while PD has a greater repertoire (e.g. reals, characters, and strings). PD specification language also has a library of generic sequences, sets, bags, and maps well-suited to ML [35]. A limitation of PD is that it discourages reference semantics [30]. It is well-known that the presence of multiple references to a common object causes aliasing and makes sound and complete static verification problematic. Therefore, PD, unlike say Java and Eiffel, adopts a value semantics by default and discourages the use of reference semantics [14]. Despite these limitations, we have adopted PD for automated deduction in our ES-Verify tool, and we are in the process of constructing a library of base Eiffel classes with a value semantics (see Introduction) using the Eiffel **expanded** construct. As a future goal we have to expand our tool to handle verification of reference aliasing and

[14] In PD, if a reference semantics is adopted, then, roughly speaking, a `heap` declaration, e.g. `heap` MyHeap, would be required. Although we have several simple PD examples on basic aliasing effect, we have not yet experienced much the power of the prover on handling reference semantics. Escher Technologies Ltd. is in the process of developing a new beta intending to properly handle the issue.

inheritance.

The theoretical foundations of PD are Floyd-Hoare logic and Dijkstra's weakest pre-condition calculus and it has the power of first-order predicate calculus, as well as a few higher-order constructs [29]. The prover generates verification conditions and aims for verifying the total correctness (termination and refinement satisfying specification) of the input code. It delivers either a proof, upon success in discharging all verification conditions, or otherwise a list of warnings, possibly accompanied by useful fix suggestions. Output from the prover can be in formats such as HTML or Tex. From an academic point of view, there is a lack of information about the inner workings of the PD theorem prover (as opposed to an interactive theorem-proving system such as *Isabelle* [17]). Ideally, the logical rules used in correctness proofs should be open for inspection so that independent trust can be established. However, the PD theorem prover does provide the complete proof, and thus the product is robust and suitable for engineering use [36].

Outline of Routine Translation:

As stated, Eiffel commands and queries become PD schemas and functions, respectively. For an Eiffel command that may modify the current object, frame constraints are needed. In order to specify frame constraints, PD supports a **change** clause[15]. For translation into PD, we use in Eiffel specification a `pd_modify`[16] declaration with its string argument passed as a list of attributes that the PD

[15] The new ECMA specification for Eiffel has a somewhat equivalent **only** clause.

[16] A boolean function that takes as argument a string and always returns true, and thus can always pass the run-time contract checking. Expression `pd_modify("*")` is an abbreviation meaning all attributes may change.

schema may change. For an Eiffel command or query, its require clause (for precondition) and ensure clause (for post-condition) appear as equivalent PD **pre** and **satisfy** clauses, respectively. For Eiffel command, its ensure clause (with its pd_modify declaration) appears as the equivalent PD change and satisfy clauses under a **post** declaration. For Eiffel query, it is translated in the same way as it for a command except there is no pd_modify declaration in its post-condition, and thus there exists no change list and post declaration for its translation in PD. Moreover, the Eiffel **old** notation for the value of expressions in a pre-state is converted into the equivalent PD primed notation. Finally, the body of an Eiffel command or query appears as an equivalent PD **via ... end** refinement segment.

C.4 Conclusion

In this chapter, we have introduced a system where we make use of the mathematical but executable ML library and the translator to convert clean and expressive Eiffel code into PD for automated verification. The translation process transforms each Eiffel construct into an equivalent PD one so that this one-to-one relation between Eiffel and PD constructs allows us to assign the semantics of the PD language to that of Eiffel. Of course such semantics depends upon the soundness of PD.

When the ES-Verify translator is applied to the Eiffel code for the birthday book example, the PD theorem prover generates 158 verification conditions which are *all* automatically discharged. This includes proof of termination via the loop variant. We used a value semantics class `ESV_ARRAY` for the two implementa-

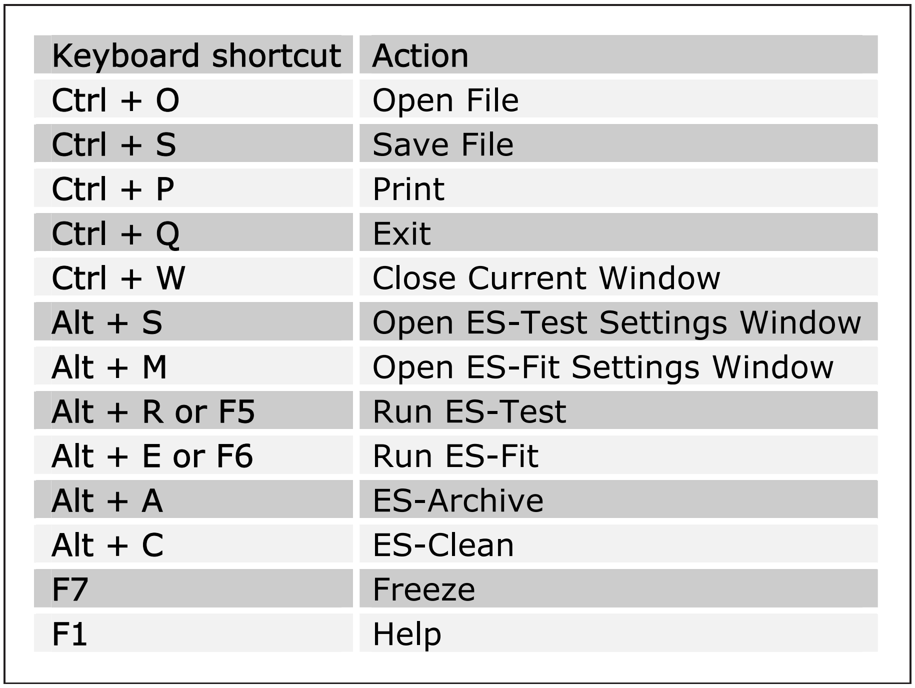
```
class  MY_STACK[G] create
   make

feature {ANY} -- public feature declaration

   make is -- constructor
      do
         create imp ; count := 0
      ensure
         pd_modify ("*")
         # model = 0 and  count = 0
      end

   count: INTEGER

   item: G is
      require count > 0
      do
         Result := imp [count - 1]
      ensure Result = model.last
      end

   put(x: G) is
      do
         if imp.is_empty then imp.force (x, 0)
         else
            if count = imp.count then
               imp.grow (imp.count * 2)
            end
            imp.put (x, count)
         end
         count := count + 1
      ensure
         pd_modify ("*")
         count = old count + 1 and then  model |=| (old model |> x)
      end

feature {ML_MODEL} -- implementation feature declaration
   imp: ESV_ARRAY[G]

feature {ML_MODEL, ANY} -- model feature declaration
   model: ML_SEQ[G] is
      do
         create Result.make; Result := Result.from_array(imp.subarray (0, count -1))
      ensure
         Result |=| Result.from_array (imp.subarray (0, count -1 ))
      end

invariant
   count >= 0 and then count <= imp.count  and then count <= # model
end
```

```
import
   "ESV_ARRAY.pd",  "ML_COLLECTION.pd";

class MY_STACK of ( G ) ^=
   abstract
      var model:  ML_SEQ of ( G ),
            count: int;
      invariant count <= #model;

   internal          //refinement
      var imp: ESV_ARRAY of ( G );
      invariant count >= 0 & count <= #imp;

      function model
         ^= (for i :: 0 .. <#imp.slice(0, (count-1)-0+1) yield imp.slice(0, (count-1)-0+1)[i]);

      function model_verification: ML_SEQ of ( G )
         ^= (for i :: 0 .. <#imp.slice(0, (count-1)-0+1) yield imp.slice(0, (count-1)-0+1)[i])
         via
            var Result:  ML_SEQ of ( G ); Result! =  ML_SEQ of ( G ){} ;
            Result! =
               (for i :: 0 .. <#imp.slice(0, (count-1)-0+1) yield imp.slice(0, (count-1)-0+1)[i]);
            value Result;
         end;

   interface        //public methods
      function count;

      build {}  //constructor equivalent to Eiffel `make'
         post
            change model, count satisfy #model' = 0 & self'.count = 0
         via imp! =  ESV_ARRAY of ( G ){} ; count! = 0 end;

      schema! put(x : G)
         post
            change
               model, count
            satisfy
               self'.count = count + 1, model' = model.append( x )
         via
            if [imp.empty]: imp! = force @ ESV_ARRAY_HELPER of  G (imp, x, 0) ;
               []:
                  if [count = #imp]:
                        imp! = grow @ ESV_ARRAY_HELPER of  G (imp, #imp*2);
                     []: pass
                  fi ; imp! = put @ ESV_ARRAY_HELPER of  G (imp, x, count)
            fi ; count! = count + 1
         end;

      function item: G
         pre count > 0 satisfy result = model.last
         via var Result:  G ; Result! = imp[count - 1] ; value Result end;

end;
```

Figure C.7: Translation Layout from Eiffel into Perfect

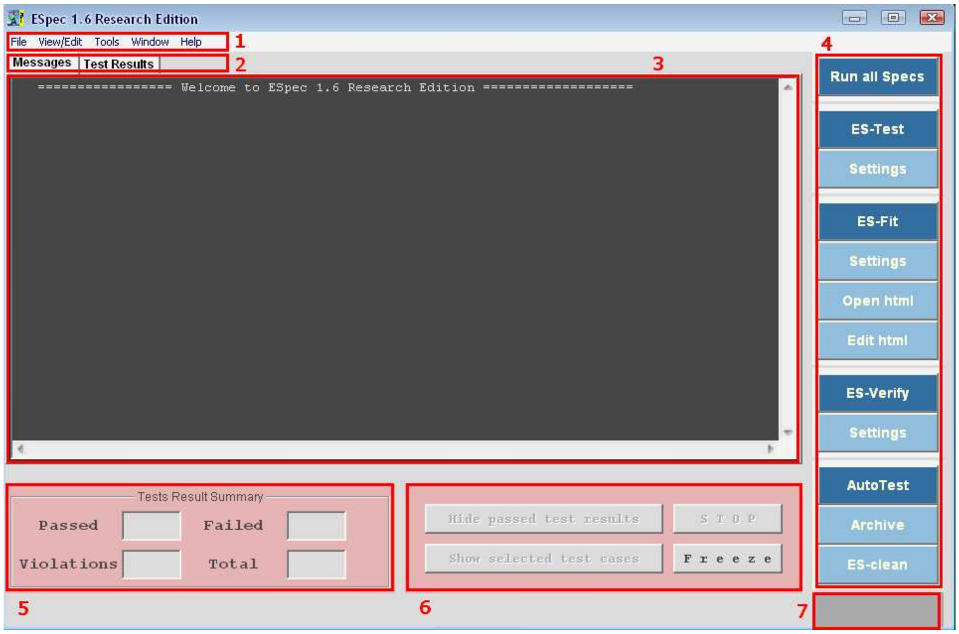
tion arrays. Preliminary experience with other examples indicates that the vast majority of verification conditions are quickly and automatically discharged, including loop variants and invariants, without any interaction with the user. The user may add axioms (with the danger of introducing inconsistencies) or assertions to help the theorem prover, but this is mostly unnecessary. Future work aims to extend the verification to handle the issue of reference aliasing and inheritance.

D Appendix: Screenshots of the ESpec Tool



Keyboard shortcut	Action
Ctrl + O	Open File
Ctrl + S	Save File
Ctrl + P	Print
Ctrl + Q	Exit
Ctrl + W	Close Current Window
Alt + S	Open ES-Test Settings Window
Alt + M	Open ES-Fit Settings Window
Alt + R or F5	Run ES-Test
Alt + E or F6	Run ES-Fit
Alt + A	ES-Archive
Alt + C	ES-Clean
F7	Freeze
F1	Help

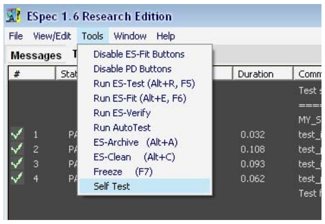
Figure D.1: ESpec shortcuts



```
================ Welcome to ESpec 1.6 Research Edition ==================
```

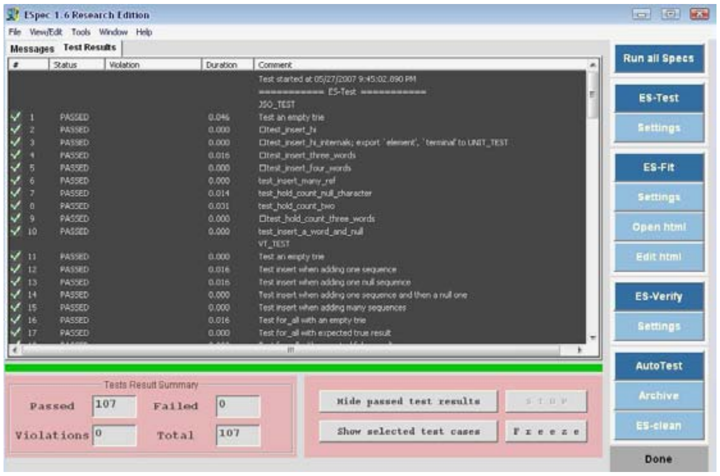
The main display of ESpec: (1) Menu bar (2) Window tabs: messages generated by ESpec is shown in "Messages" tab and test results are shown in "Test Results" tab, files are open in separate tabs for editing (3) Editor window: shows the file contents to the user and allows the user to edit the contents or select the test results (4) Command buttons: "Run all Specs" button executes all tests (i.e., Unit tests, Fit tests and ES-Verify) at the same time. "ES-Test", "ES-Fit" and "ES-Verify" buttons only run ES-Test, ES-Fit and ES-Verify respectively. "Settings" buttons are used for user settings. "Open html" opens the HTML document selected by the user (user selects the file on the Editor window) (5) Tests results summary box: shows a summary of the test results to the user. Passed is the number of tests (Unit, Fit) that are passed, Failed is the number of tests that failed (Unit or Fit or verification modules), Violations is the number of Contract violations that happen during the test (Unit, Fit), Total is the total number of test cases executed (6) User buttons: "Freeze" button is used to re-compile the system when user made some changes to the test case. "Stop" button kills the running processes and stops the testing process (7) Progress bar: shows the status of the tool

Figure D.2: ESpec main window



```
#    Stat                 Duration   Comm
                                     Test s
                                     ====
                                     MY_S
✓ 1  PA                   0.032      test_
✓ 2  PA                   0.108      test_
✓ 3  PA                   0.093      test_
✓ 4  PA                   0.062      test_
                                     Test f
```

ESpec self-test: ESpec provides an option called "Self test" to test itself. It is recommended that user runs the "Self Test" tool the first time ESpec is installed. This makes sure that ESpec is installed and works correctly. To do a self test, select "Self Test" from the "Tools" menu.



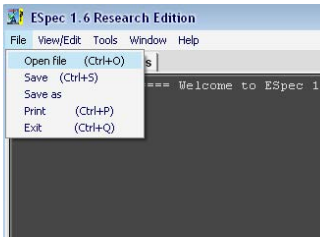
```
#      Status    Violation   Duration   Comment
                                        Test started at 05/27/2007 9:45:02.890 PM
                                        =========== ES-Test ===========
                                        JSO_TEST
✓ 1    PASSED                0.046      Test an empty trie
✓ 2    PASSED                0.000      □test_insert_hi
✓ 3    PASSED                0.000      □test_insert_hi_internals; export `element', `terminal' to UNIT_TEST
✓ 4    PASSED                0.016      □test_insert_three_words
✓ 5    PASSED                0.000      □test_insert_four_words
✓ 6    PASSED                0.000      test_insert_many_ref
✓ 7    PASSED                0.014      test_hold_count_null_character
✓ 8    PASSED                0.031      test_hold_count_two
✓ 9    PASSED                0.000      □test_hold_count_three_words
✓ 10   PASSED                0.000      test_insert_a_word_and_null
                                        VT_TEST
✓ 11   PASSED                0.000      Test an empty trie
✓ 12   PASSED                0.016      Test insert when adding one sequence
✓ 13   PASSED                0.016      Test insert when adding one null sequence
✓ 14   PASSED                0.000      Test insert when adding one sequence and then a null one
✓ 15   PASSED                0.000      Test insert when adding many sequences
✓ 16   PASSED                0.016      Test for_all with an empty trie
✓ 17   PASSED                0.000      Test for_all with expected true result
```

Tests Result Summary

Passed	107	Failed	0
Violations	0	Total	107

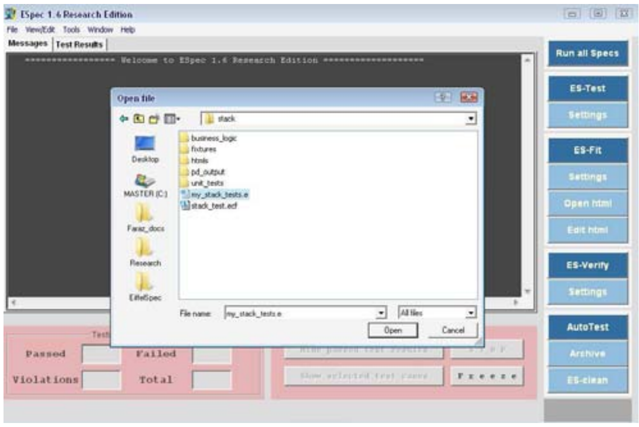
Self Test result: The result of self test will be shown on ESpec's main window. A green bar indicates that ESpec is installed correctly. Please note: during the first execution of ESpec, a firewall warning may be generated. User must allow ESpec to access local sockets by pressing "Allow" or "Unblock".

Figure D.3: Self test option



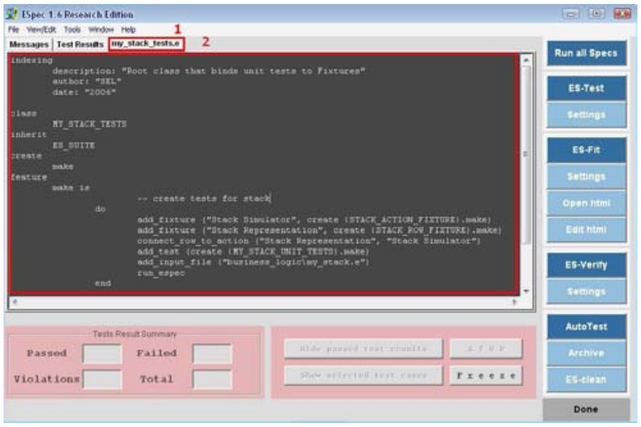
```
=== Welcome to ESpec 1
```

Open File: ESpec allows users to open and edit any text file (e.g., *.e files). Select the "Open File" from the File menu and choose the name of the file.



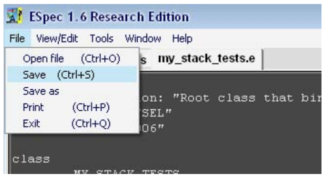
```
================= Welcome to ESpec 1.6 Research Edition ==================
```

(1) Editor tab: each file will open in a new editor tab (2) User can directly edit the file.



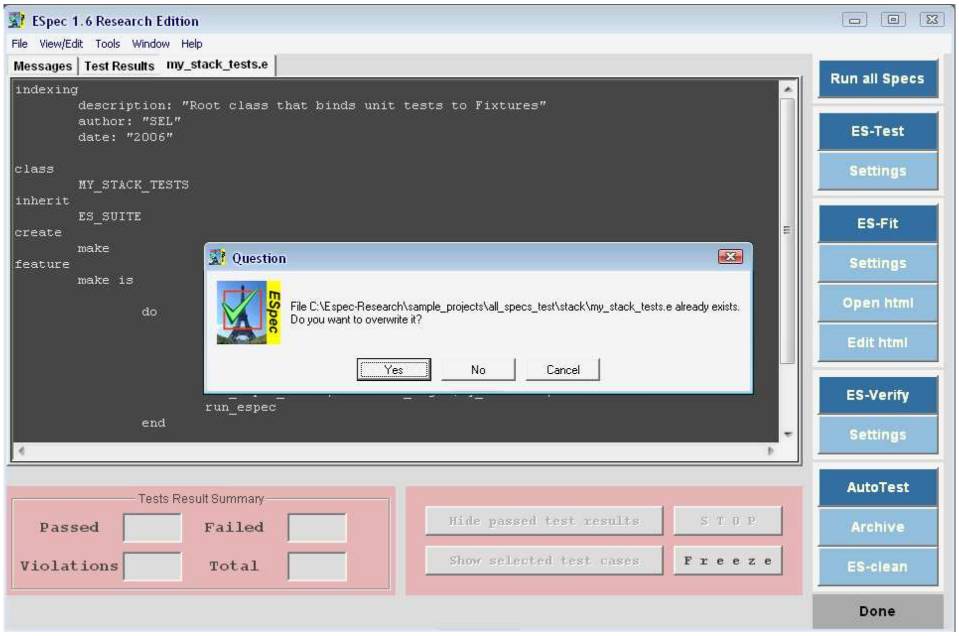
```
indexing
        description: "Root class that binds unit tests to Fixtures"
        author: "SEL"
        date: "2006"

class
        MY_STACK_TESTS
inherit
        ES_SUITE
create
        make
feature
        make is
                        -- create tests for stack
                do
                        add_fixture ("Stack Simulator", create {STACK_ACTION_FIXTURE}.make)
                        add_fixture ("Stack Representation", create {STACK_ROW_FIXTURE}.make)
                        connect_row_to_action ("Stack Representation", "Stack Simulator")
                        add_test (create {MY_STACK_UNIT_TESTS}.make)
                        add_input_file ("business_logic\my_stack.e")
                        run_espec
                end
```

Figure D.4: Opening a file



```
on: "Root class that bi
SEL"
06"

class
```

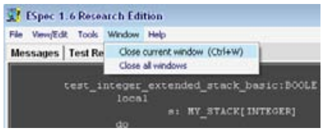
Save File: Users can directly edit their opened files in the Editor Window. To save the file, select "Save" from the menu items.



```
indexing
        description: "Root class that binds unit tests to Fixtures"
        author: "SEL"
        date: "2006"

class
        MY_STACK_TESTS
inherit
        ES_SUITE
create
        make
feature
        make is
                do

                        run_espec
                end
```

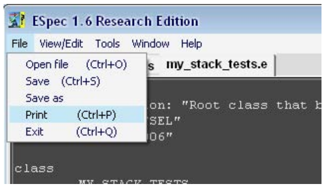
File C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack_tests.e already exists.
Do you want to overwrite it?

Close window: After editing a file, user may close the tab by selecting "close current window" from the "Window" menu.



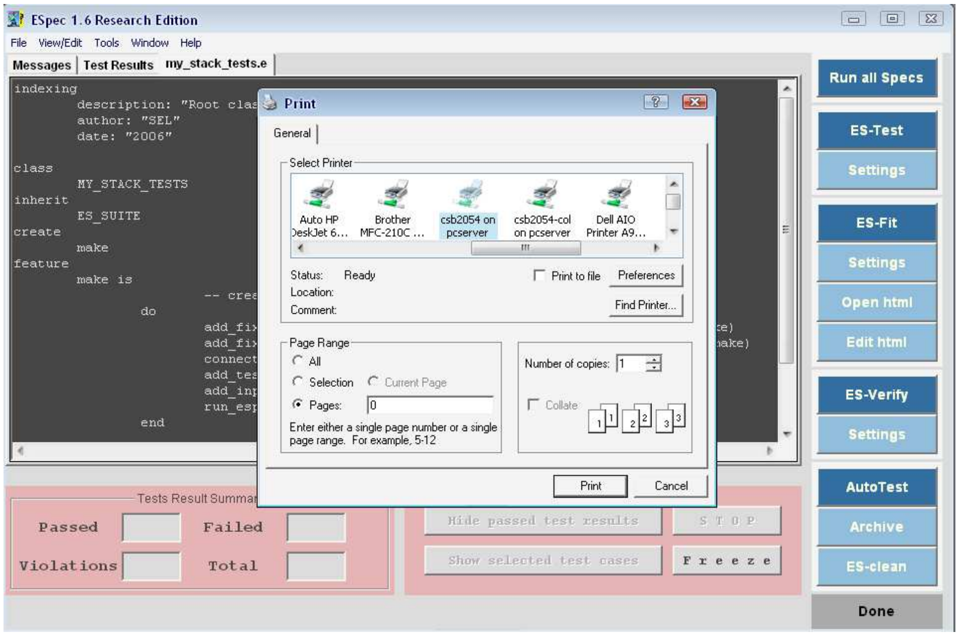
```
        test_integer_extended_stack_basic:BOOLE
                local
                        s: MY_STACK[INTEGER]
                do
```

Figure D.5: Editing, Saving and closing a file



```
on: "Root class that b
SEL"
06"

class
    MY_STACK_TESTS
```

Print File: To print a file to the printer, select the "print" menu item.



```
indexing
        description: "Root clas
        author: "SEL"
        date: "2006"

class
        MY_STACK_TESTS
inherit
        ES_SUITE
create
        make
feature
        make is
                                -- crea
                do
                                add_fi
                                add_fi
                                connec
                                add_te
                                add_in
                                run_es
                end
```

Status: Ready

Location:

Comment:

○ All

○ Selection ○ Current Page

● Pages: 0

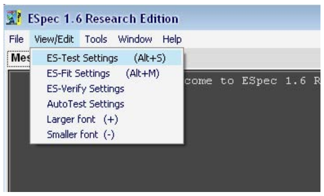
Enter either a single page number or a single page range. For example, 5-12

Number of copies: 1

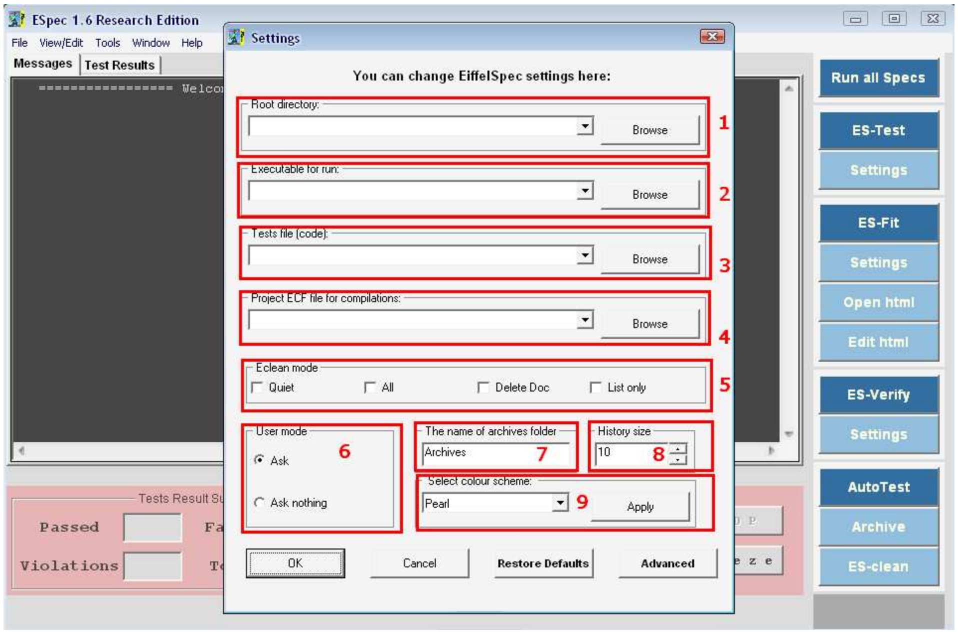
□ Print to file

□ Collate

Figure D.6: Printing a File



Setting up the project: Before running the tests, user should setup the system under test using the settings window. This window can be opened by choosing the "ES-Test Settings" from the menu items under "View/Edit" menu bar or alternatively by pressing the "Settings" button under "ES-Test" button.



You can change EiffelSpec settings here:

Root directory:

Executable for run:

Tests file (code):

Project ECF file for compilations:

Eclean mode: ☐ Quiet ☐ All ☐ Delete Doc ☐ List only

User mode: ● Ask ○ Ask nothing

The name of archives folder: Archives

History size: 10

Select colour scheme: Pearl

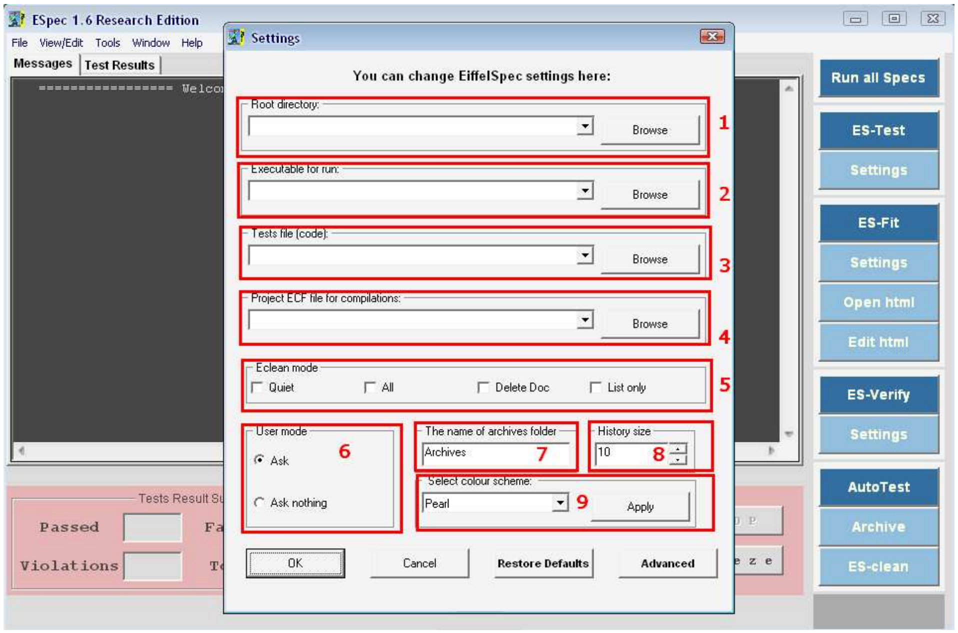
Tests Result Su

Passed

Violations

Setting options: (1) Root directory: browse to the directory of the system under test. This directory will be used by ES-Clean and ES-Archive (optional) (2) Choose the workbench executable file generated by the compiler (located in EIFGEN folder under W_code). This file will be executed every time the tool is invoked. (3) A unit test file can be preset in here for further editing and re-compiling (optional) (4) Choose the ECF file for the project. This file will be used for Freezing the system after each change.

Figure D.7: ESpec settings



You can change EiffelSpec settings here:

Root directory:

Executable for run:

Tests file (code):

Project ECF file for compilations:

Eclean mode: □ Quiet □ All □ Delete Doc □ List only

User mode: ● Ask ○ Ask nothing

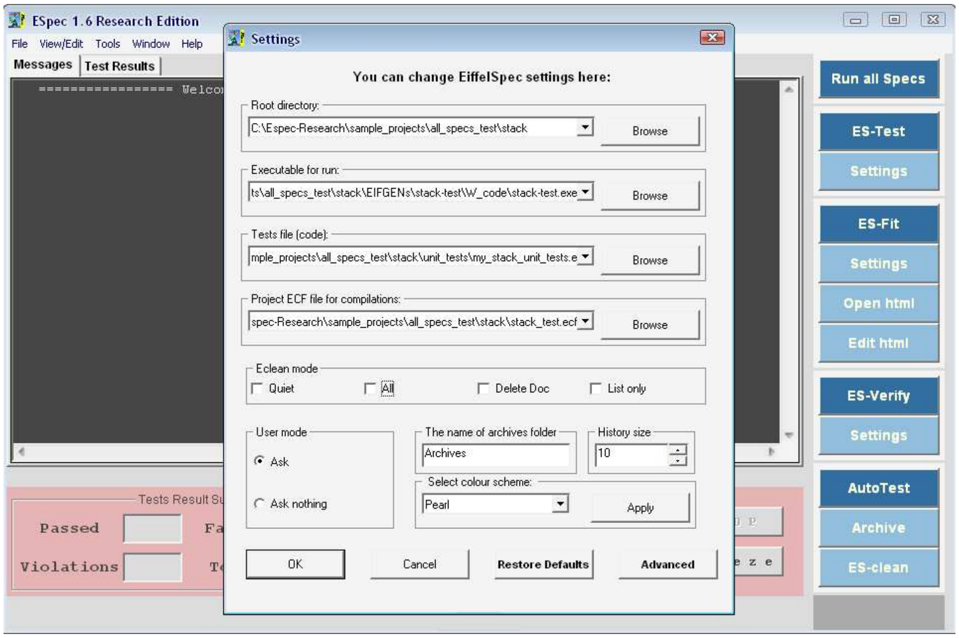
The name of archives folder: Archives

History size: 10

Select colour scheme: Pearl

Setting up the project continued: (5) ES-Clean removes the compiler generated files (in EIFGEN folder) and it could be invoked by pressing the "ES-Clean" command button from the GUI. ES-Clean has different modes: (a) Quiet (does not show the list of files that has been deleted), (b) All: removes all automatically generated files such as Document and Diagram. (c) DeleteDoc: removes the Document folder. (d) List only: shows the user the list of files to be deleted without actually deleting them. (6) User mode: user can disable re-confirmations of ESpec by selecting "Don't ask". (7) Name of the archive folder can be set here. (8) User can change the size of the history list. (9) Appearance of the ESpec.

Figure D.8: ESpec settings continued



You can change EiffelSpec settings here:

Root directory: C:\Espec-Research\sample_projects\all_specs_test\stack

Executable for run: ts\all_specs_test\stack\EIFGENs\stack-test\W_code\stack-test.exe

Tests file (code): mple_projects\all_specs_test\stack\unit_tests\my_stack_unit_tests.e

Project ECF file for compilations: spec-Research\sample_projects\all_specs_test\stack\stack_test.ecf

Eclean mode: □ Quiet □ All □ Delete Doc □ List only

User mode: ● Ask ○ Ask nothing

The name of archives folder: Archives

History size: 10

Select colour scheme: Pearl

Running the unit tests: After setting up the project using "Settings Windows", it is time to run the unit test. The unit tests can be executed by pressing the "ES-Test" command button on the GUI or selecting the "Run ES-Test" from the menu item.

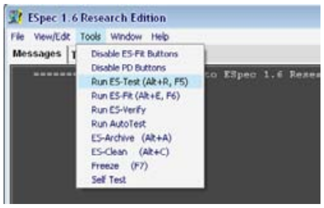
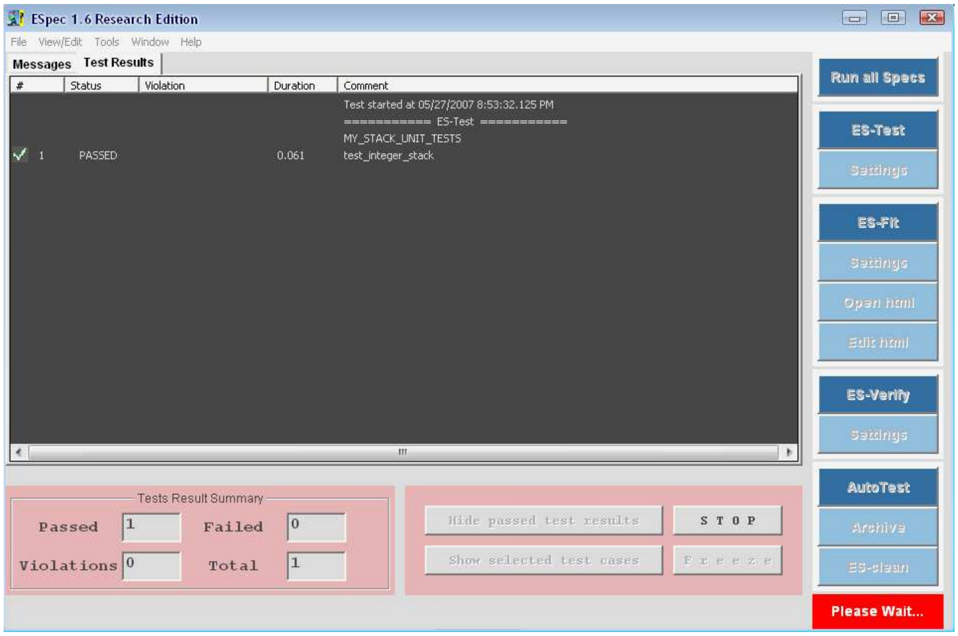


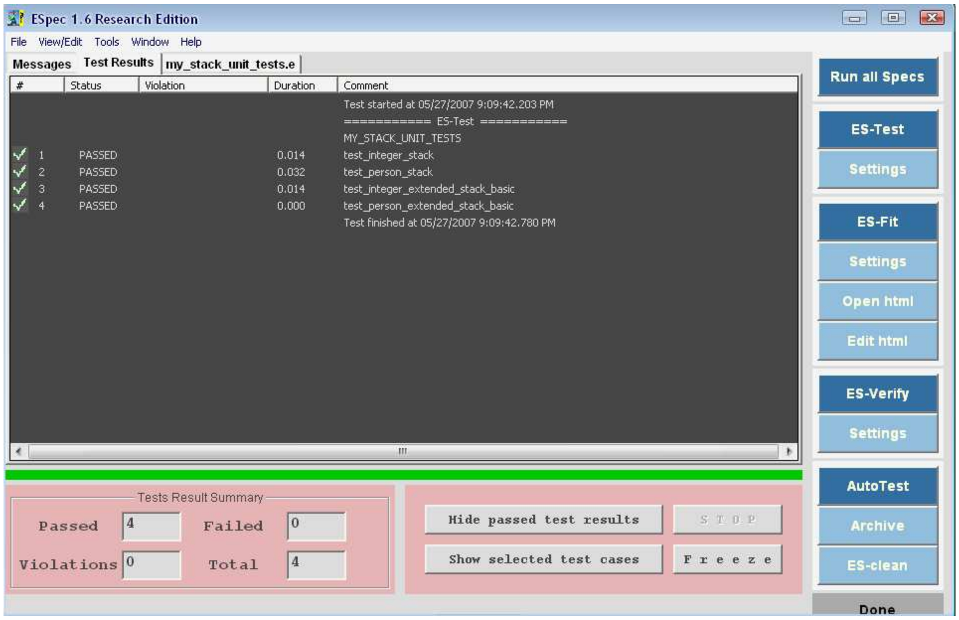
Figure D.9: Run ES-Test



```
#   Status   Violation   Duration   Comment
                                    Test started at 05/27/2007 8:53:32.125 PM
                                    =========== ES-Test ===========
                                    MY_STACK_UNIT_TESTS
✓ 1 PASSED               0.061      test_integer_stack
```

Passed 1 Failed 0
Violations 0 Total 1

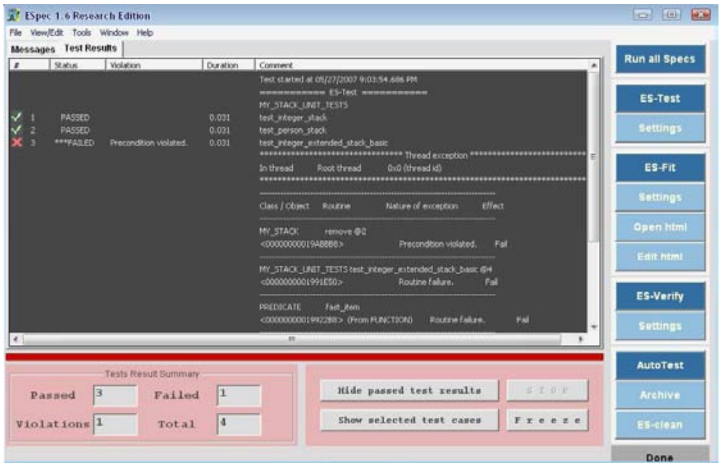
ES-Test in progress: When the "ES-Test" is invoked and is running, the ESpec status will be changed to "Please Wait". User can interrupt execution of the tests by pressing the "Stop" button at any time.



```
#   Status   Violation   Duration   Comment
                                    Test started at 05/27/2007 9:09:42.203 PM
                                    =========== ES-Test ===========
                                    MY_STACK_UNIT_TESTS
✓ 1 PASSED               0.014      test_integer_stack
✓ 2 PASSED               0.032      test_person_stack
✓ 3 PASSED               0.014      test_integer_extended_stack_basic
✓ 4 PASSED               0.000      test_person_extended_stack_basic
                                    Test finished at 05/27/2007 9:09:42.780 PM
```

Passed 4 Failed 0
Violations 0 Total 4

Figure D.10: ES-Test in progress and results

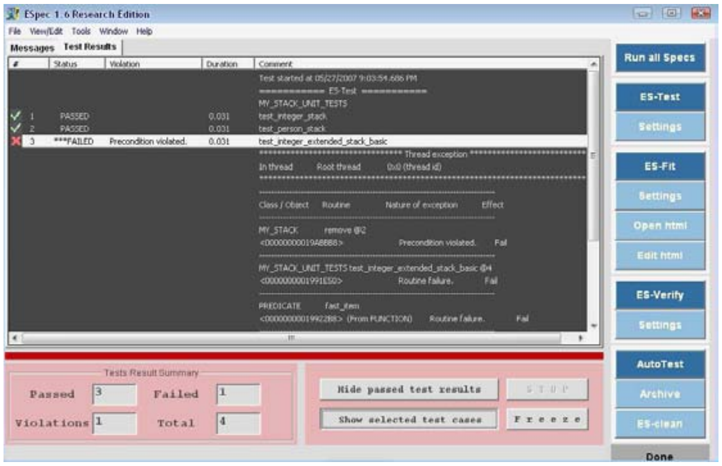


```
#     Status      Violation               Duration   Comment
                                                     Test started at 05/27/2007 9:03:54.686 PM
                                                     ============ ES-Test ============
                                                     MY_STACK_UNIT_TESTS
✓ 1    PASSED                             0.031      test_integer_stack
✓ 2    PASSED                             0.031      test_person_stack
✗ 3   ***FAILED   Precondition violated.  0.031      test_integer_extended_stack_basic
                                                     ******************************** Thread exception ************************
                                                     In thread      Root thread      0x0 (thread id)
                                                     **************************************************************************
                                                     -------------------------------------------------------------
                                                     Class / Object   Routine        Nature of exception     Effect
                                                     -------------------------------------------------------------
                                                     MY_STACK         remove @2
                                                     <00000000019ABBB8>              Precondition violated.  Fail
                                                     -------------------------------------------------------------
                                                     MY_STACK_UNIT_TESTS test_integer_extended_stack_basic @4
                                                     <0000000001991E50>              Routine failure.        Fail
                                                     -------------------------------------------------------------
                                                     PREDICATE        fast_item
                                                     <00000000001992288> (From FUNCTION)   Routine failure.   Fail
```

Tests Result Summary

Passed	3	Failed	1
Violations	1	Total	4

Error in the test results: Passed test cases are check marked in green (first column). For convenience the status of each test case (passed or failed) is reported in the second column. The failed cases (if any) are marked in red. The type of the contract violation (of any) is reported in the third column. Forth column shows the duration of execution of the test case. More information about the violation is reported to the fifth column; this option is enabled by using "`show_errors`" command in the code.



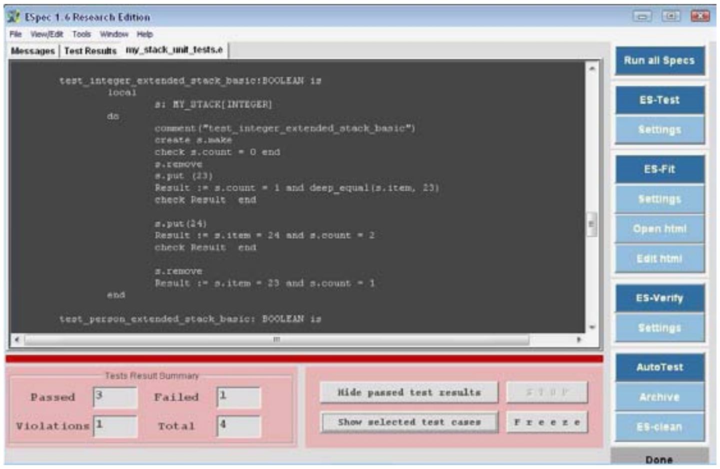
```
#     Status      Violation               Duration   Comment
                                                     Test started at 05/27/2007 9:03:54.686 PM
                                                     ============ ES-Test ============
                                                     MY_STACK_UNIT_TESTS
✓ 1    PASSED                             0.031      test_integer_stack
✓ 2    PASSED                             0.031      test_person_stack
✗ 3   ***FAILED   Precondition violated.  0.031      test_integer_extended_stack_basic
                                                     ******************************** Thread exception ************************
                                                     In thread      Root thread      0x0 (thread id)
                                                     **************************************************************************
                                                     -------------------------------------------------------------
                                                     Class / Object   Routine        Nature of exception     Effect
                                                     -------------------------------------------------------------
                                                     MY_STACK         remove @2
                                                     <00000000019ABBB8>              Precondition violated.  Fail
                                                     -------------------------------------------------------------
                                                     MY_STACK_UNIT_TESTS test_integer_extended_stack_basic @4
                                                     <0000000001991E50>              Routine failure.        Fail
                                                     -------------------------------------------------------------
                                                     PREDICATE        fast_item
                                                     <00000000001992288> (From FUNCTION)   Routine failure.   Fail
```

Tests Result Summary

Passed	3	Failed	1
Violations	1	Total	4

Selecting test cases: Test cases can be selected (using the control key + left key of mouse). To view or edit the selected test case, press the "show selected test cases" button.

Figure D.11: Failed cases and selecting test cases

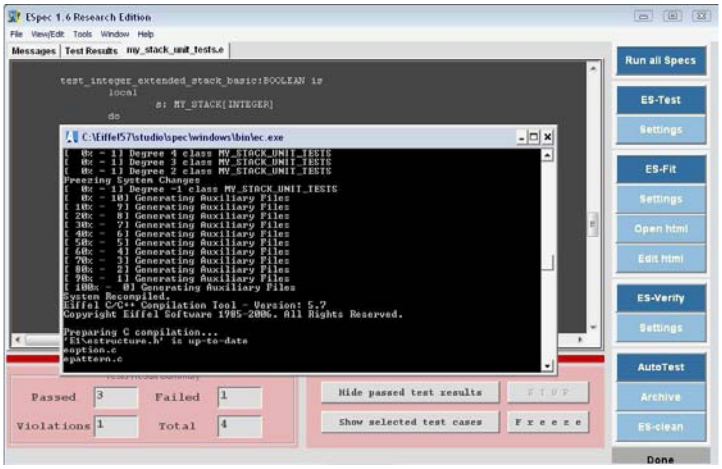


```
test_integer_extended_stack_basic:BOOLEAN is
        local
                s: MY_STACK[INTEGER]
        do
                comment("test_integer_extended_stack_basic")
                create s.make
                check s.count = 0 end
                s.remove
                s.put (23)
                Result := s.count = 1 and deep_equal(s.item, 23)
                check Result  end

                s.put(24)
                Result := s.item = 24 and s.count = 2
                check Result  end

                s.remove
                Result := s.item = 23 and s.count = 1
        end

test_person_extended_stack_basic: BOOLEAN is
```

Passed 3 Failed 1

Violations 1 Total 4

Editing the test case: When a test case is selected for editing, ESpec will open it in a new tab and allows the user to edit and save the test case directly from ESpec (saving is done as before).



```
test_integer_extended_stack_basic:BOOLEAN is
        local
                s: MY_STACK[INTEGER]
        do
```

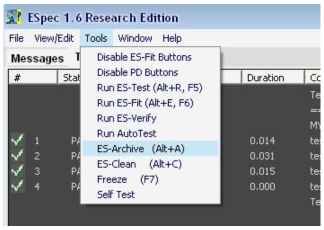
```
[  0% - 1] Degree 4 class MY_STACK_UNIT_TESTS
[  0% - 1] Degree 3 class MY_STACK_UNIT_TESTS
[  0% - 1] Degree 2 class MY_STACK_UNIT_TESTS
Freezing System Changes
[  0% - 1] Degree -1 class MY_STACK_UNIT_TESTS
[  0% - 10] Generating Auxiliary Files
[ 10% -  9] Generating Auxiliary Files
[ 20% -  8] Generating Auxiliary Files
[ 30% -  7] Generating Auxiliary Files
[ 40% -  6] Generating Auxiliary Files
[ 50% -  5] Generating Auxiliary Files
[ 60% -  4] Generating Auxiliary Files
[ 70% -  3] Generating Auxiliary Files
[ 80% -  2] Generating Auxiliary Files
[ 90% -  1] Generating Auxiliary Files
[ 100% -  0] Generating Auxiliary Files
System Recompiled.
Eiffel C/C++ Compilation Tool - Version: 5.7
Copyright Eiffel Software 1985-2006. All Rights Reserved.

Preparing C compilation...
'E1\estructure.h' is up-to-date
eoption.c
epattern.c
```

Passed 3 Failed 1

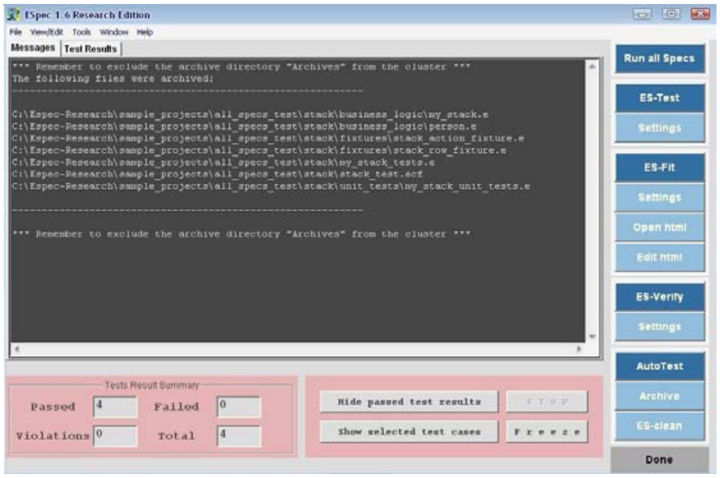
Violations 1 Total 4

Freezing the system: When there is a change in the system under test, it must be re-compiled. This can be done from ESpec GUI by pressing the "Freeze" button (Freeze assumes that the ECF configuration file is already set from the "Settings" Window).

Figure D.12: Editing test cases directly from ESpec and freezing the system



ES-Archive for backing up the project: User can archive the current project at any time by selecting the "ES-Archive" item from the "Tools" menu. ES-Archive tool will generate a backup folder in the current directory which is tagged with the time and date of this archive. EIFGENs and automatically generated (e.g., Document and Diagram) directories will not be archived.



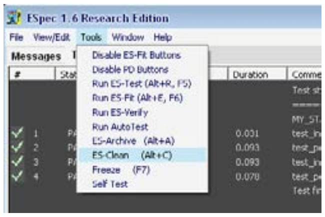
```
*** Remember to exclude the archive directory "Archives" from the cluster ***
The following files were archived:
-------------------------------------------------------------

C:\Espec-Research\sample_projects\all_specs_test\stack\business_logic\my_stack.e
C:\Espec-Research\sample_projects\all_specs_test\stack\business_logic\person.e
C:\Espec-Research\sample_projects\all_specs_test\stack\fixtures\stack_action_fixture.e
C:\Espec-Research\sample_projects\all_specs_test\stack\fixtures\stack_row_fixture.e
C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack_tests.e
C:\Espec-Research\sample_projects\all_specs_test\stack\stack_test.ecf
C:\Espec-Research\sample_projects\all_specs_test\stack\unit_tests\my_stack_unit_tests.e

-------------------------------------------------------------

*** Remember to exclude the archive directory "Archives" from the cluster ***
```

Tests Result Summary

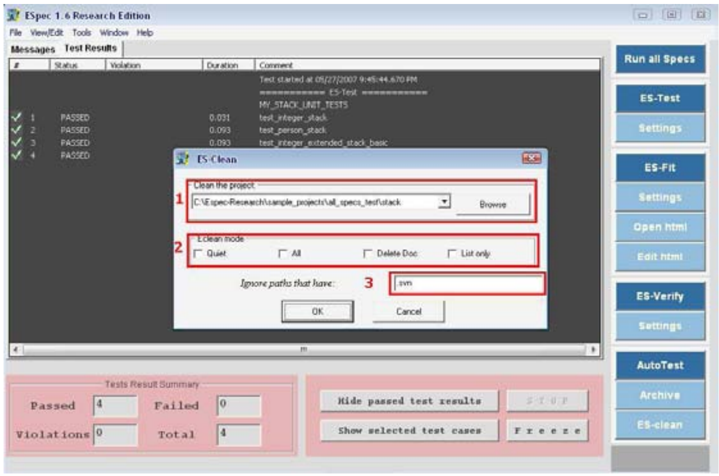
Passed	4	Failed	0
Violations	0	Total	4

ES-Archive results: ES-Archive shows the list of files which were archived at the end of the process.

Figure D.13: ES-Archive tool



ES-Clean for cleaning auto-generated files: User can remove Eiffel's automatically generated files such as files Diagrams, Documentation and EIFGENs (compiler generated files). It is recommended that users ES-Clean their projects after number of compilations. ES-Clean tool can be invoked by selecting "ES-Clean" item from the "Tools" menu.



```
#   Status   Violation   Duration   Comment
                                    Test started at 05/27/2007 9:45:44.670 PM
                                    =========== ES-Test ===========
                                    MY_STACK_UNIT_TESTS
✓ 1 PASSED               0.031      test_integer_stack
✓ 2 PASSED               0.093      test_person_stack
✓ 3 PASSED               0.093      test_integer_extended_stack_basic
✓ 4 PASSED
```

1 Clean the project: C:\Espec-Research\sample_projects\all_specs_test\stack

2 Eclean mode: □ Quiet □ All □ Delete Doc □ List only

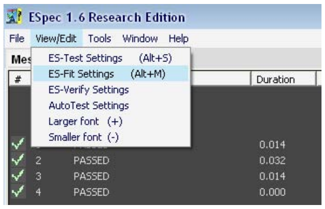
Ignore paths that have: 3 .svn

Tests Result Summary

Passed	4	Failed	0
Violations	0	Total	4

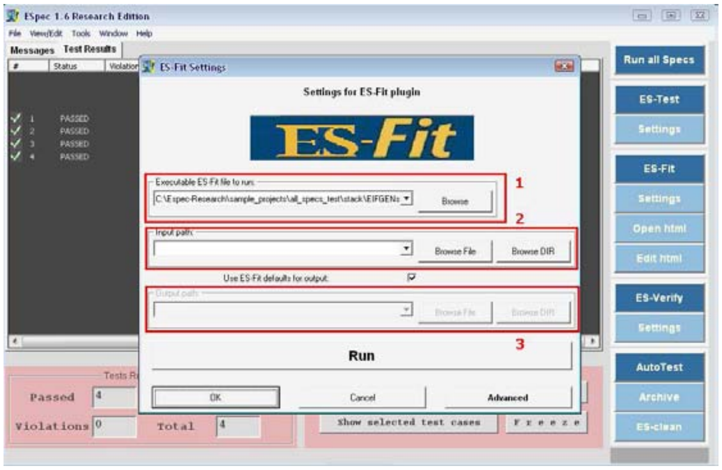
ES-Clean window: (1) User can select which project will be ES-Cleaned: by default the current project is selected, however ES-pec allows the user to choose another project. (2) User can select ES-Clean mode (Quiet: no reports will be generated but files will be removed, All: EIFGENs, Documents and Diagram files will be deleted. DeleteDoc: remove the generated documentation files (files in Document folder), List only: only shows the list of files to-be-deleted (no files are actually removed). (3) type of files to be ignored by ES-Clean.

Figure D.14: ES-Clean tool



```
✓        0.014
✓ 2  PASSED  0.032
✓ 3  PASSED  0.014
✓ 4  PASSED  0.000
```

ES-Fit settings: In order to execute the ES-Fit tool, user needs to first initialize the ES-Fit settings. User can open the settings window by selecting "ES-Fit Settings" item from the "Tools" menu.



```
✓ 1  PASSED
✓ 2  PASSED
✓ 3  PASSED
✓ 4  PASSED
```

Executable ES-Fit file to run: C:\Espec-Research\sample_projects\all_specs_test\stack\EIFGENs

Input path:

Use ES-Fit defaults for output: ☑

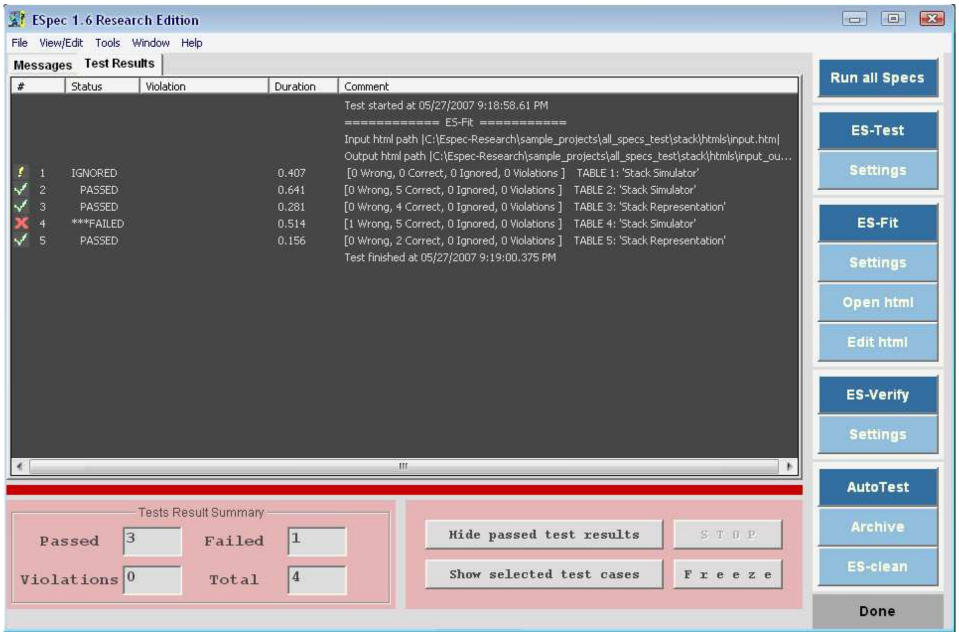
Passed 4

Violations 0

Total 4

Setting ES-Fit inputs: (1) Executable ES-Fit file to run: this box is the project executable file (default is the current executable) (2) Input path: is the path of the input HTML requirement document. User can either select a single HTML or HTM document or a directory containing many HTML files. (3) Output path: is the path where output files are generated. By default it is going to be the same path as the input files.

Figure D.15: ES-Fit settings



#	Status	Violation	Duration	Comment
				Test started at 05/27/2007 9:18:58.61 PM
				============= ES-Fit =============
				Input html path |C:\Espec-Research\sample_projects\all_specs_test\stack\htmls\input.html|
				Output html path |C:\Espec-Research\sample_projects\all_specs_test\stack\htmls\input_ou...
! 1	IGNORED		0.407	[0 Wrong, 0 Correct, 0 Ignored, 0 Violations] TABLE 1: 'Stack Simulator'
✓ 2	PASSED		0.641	[0 Wrong, 5 Correct, 0 Ignored, 0 Violations] TABLE 2: 'Stack Simulator'
✓ 3	PASSED		0.281	[0 Wrong, 4 Correct, 0 Ignored, 0 Violations] TABLE 3: 'Stack Representation'
✗ 4	***FAILED		0.514	[1 Wrong, 5 Correct, 0 Ignored, 0 Violations] TABLE 4: 'Stack Simulator'
✓ 5	PASSED		0.156	[0 Wrong, 2 Correct, 0 Ignored, 0 Violations] TABLE 5: 'Stack Representation'
				Test finished at 05/27/2007 9:19:00.375 PM

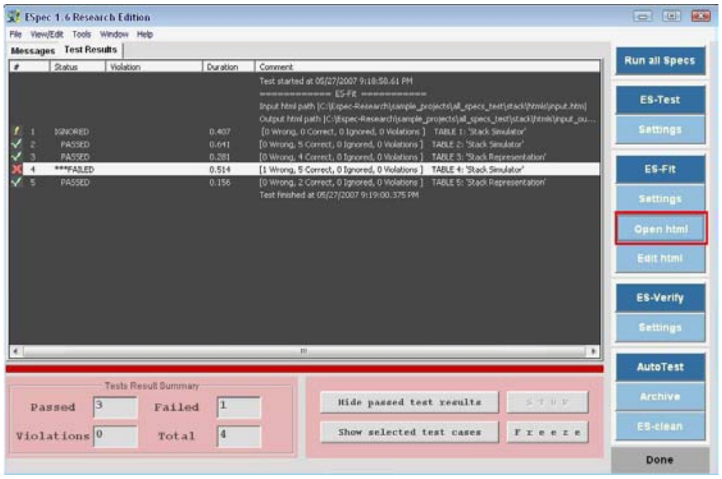
Tests Result Summary

Passed 3 Failed 1

Violations 0 Total 4

ES-Fit execution: After setting up the initial settings, user can execute the Fit tables (specified in the input path) by pressing the "Run" button in the "ES-Fit settings" window or simply by pressing "ES-Fit" command button. Similar to ES-Test the ES-Fit results are displayed in the main ESpec window. Each table in the HTML input document is treated as a single case. A table that does not have any failure (all rows are passed) will be shown as a passed table (with green checkmark in the first column). A failed table is a table that has at least one failed row which is marked as red. Tables can be ignored (in case of a reference table or tables with heading containing the word "Ignore") which will be shown in yellow. A detailed report for each table is generated in the fifth column of the display in the form of [p Wrong, q Correct, r Ignored, s Violations]. where p, q, r, and s are the number of cells in the table which are failed, passed, ignored or generated violations respectively.

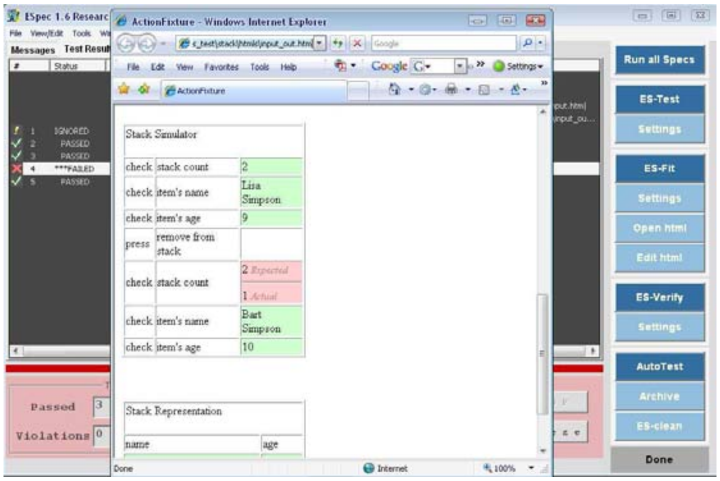
Figure D.16: ES-Fit execution



```
#   Status      Violation   Duration   Comment
                                       Test started at 05/27/2007 9:18:58.61 PM
                                       ============ ES-Fit ============
                                       Input html path [C:\Espec-Research\sample_projects\all_specs_test\stack\html\input.html]
                                       Output html path [C:\Espec-Research\sample_projects\all_specs_test\stack\html\input_ou...
1   IGNORED                 0.407      [0 Wrong, 0 Correct, 0 Ignored, 0 Violations ]   TABLE 1: 'Stack Simulator'
2   PASSED                  0.641      [0 Wrong, 5 Correct, 0 Ignored, 0 Violations ]   TABLE 2: 'Stack Simulator'
3   PASSED                  0.281      [0 Wrong, 4 Correct, 0 Ignored, 0 Violations ]   TABLE 3: 'Stack Representation'
4   ***FAILED               0.514      [1 Wrong, 5 Correct, 0 Ignored, 0 Violations ]   TABLE 4: 'Stack Simulator'
5   PASSED                  0.156      [0 Wrong, 2 Correct, 0 Ignored, 0 Violations ]   TABLE 5: 'Stack Representation'
                                       Test finished at 05/27/2007 9:19:00.375 PM
```

Passed 3 Failed 1
Violations 0 Total 4

Opening HTML documents: To see the generated HTML (the ES-Fit result) or the input HTML document, user can select the entry in the ESpec display corresponding to a Fit table, and press "Open html" command button.



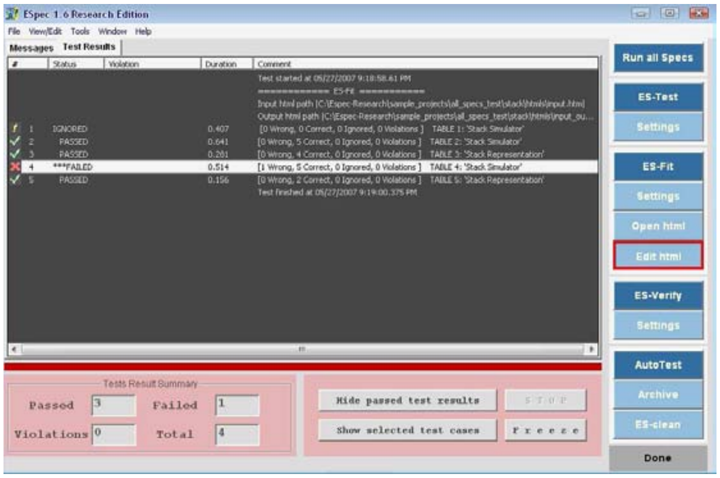
ActionFixture - Windows Internet Explorer
s_test\stack\html\input_out.html
File Edit View Favorites Tools Help
Google
Settings
ActionFixture
Done
Internet
100%

Stack Simulator		
check	stack count	2
check	item's name	Lisa Simpson
check	item's age	9
press	remove from stack	
check	stack count	2 *Expected* 1 *Actual*
check	item's name	Bart Simpson
check	item's age	10

Stack Representation	
name	age

Viewing the HTML documents: After pressing the "Open html" button, the selected Fit document will open in the browser (or ESpec's internal HTML editor—shown in the sequel).

Figure D.17: Opening HTML in ES-Fit

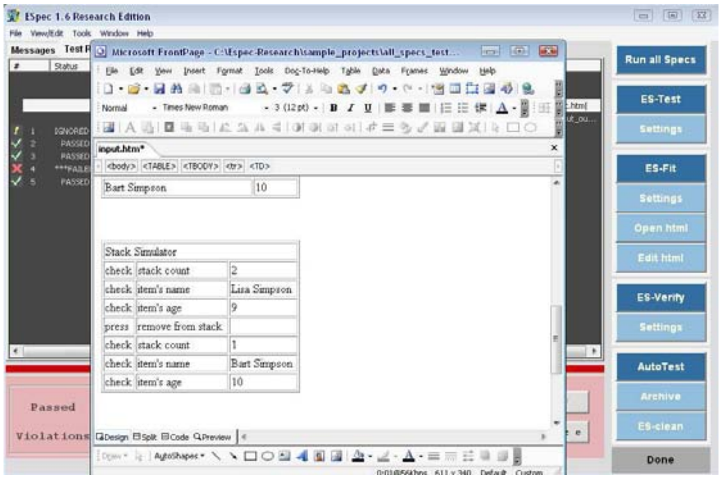


#	Status	Violation	Duration	Comment
				Test started at 05/27/2007 9:18:58.61 PM
				============= ES-Fit ============
				Input html path [C:\Espec-Research\sample_projects\all_specs_test\stack\htmls\input.htm]
				Output html path [C:\Espec-Research\sample_projects\all_specs_test\stack\htmls\input_ou...
1	IGNORED		0.407	[0 Wrong, 0 Correct, 0 Ignored, 0 Violations] TABLE 1: 'Stack Simulator'
2	PASSED		0.641	[0 Wrong, 5 Correct, 0 Ignored, 0 Violations] TABLE 2: 'Stack Simulator'
3	PASSED		0.281	[0 Wrong, 4 Correct, 0 Ignored, 0 Violations] TABLE 3: 'Stack Representation'
4	***FAILED		0.514	[1 Wrong, 5 Correct, 0 Ignored, 0 Violations] TABLE 4: 'Stack Simulator'
5	PASSED		0.156	[0 Wrong, 2 Correct, 0 Ignored, 0 Violations] TABLE 5: 'Stack Representation'
				Test finished at 05/27/2007 9:19:00.375 PM

Passed 3 Failed 1

Violations 0 Total 4

Editing HTML documents: User can also directly edit the input HTML document from the ESpec tool by selecting the Fit table entry and pressing the "Edit html" button.

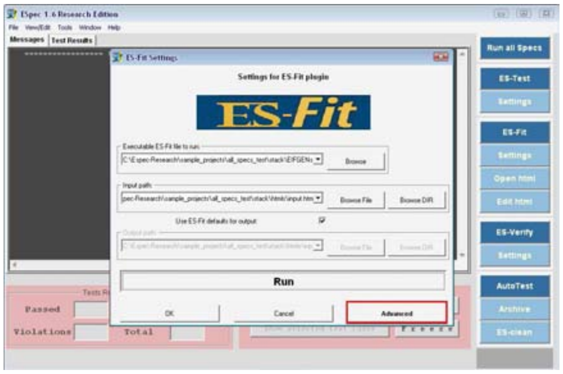

ESpec 1.6 Research Edition
File View/Edit Tools Window Help
Microsoft FrontPage - C:\Espec-Research\sample_projects\all_specs_test...
File Edit View Insert Format Tools Doc-To-Help Table Data Frames Window Help
Normal
Times New Roman
3 (12 pt)
input.htm*
<body> <TABLE> <TBODY> <tr> <TD>
Design Split Code Preview
Draw
AutoShapes
Run all Specs
ES-Test
Settings
ES-Fit
Settings
Open html
Edit html
ES-Verify
Settings
AutoTest
Archive
ES-clean
Done
Passed
Violations

Bart Simpson	10

Stack Simulator		
check	stack count	2
check	item's name	Lisa Simpson
check	item's age	9
press	remove from stack	
check	stack count	1
check	item's name	Bart Simpson
check	item's age	10

Editing in FrontPage: After pressing the "Edit html" button, the selected Fit document will open in the FrontPage (or ESpec's internal HTML editor—shown in the sequel).

Figure D.18: Editing HTML in ES-Fit



Settings for ES-Fit plugin

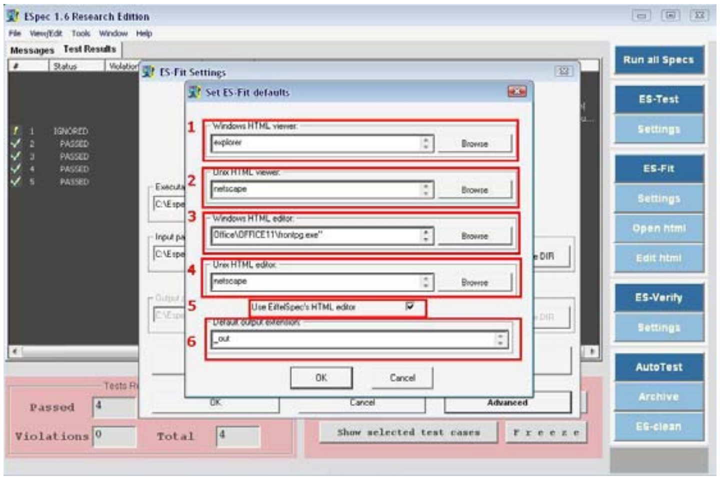
Executable ES-Fit file to run: C:\Espec-Research\sample_projects\all_specs_test\stack\EIFGENs

Input path: pec-Research\sample_projects\all_specs_test\stack\html\input.htm

Use ES-Fit defaults for output ☑

Output path: C:\Espec-Research\sample_projects\all_specs_test\stack\html\inp

ES-Fit advanced settings: Depending on the user preference, any HTML editor tool can be used for opening/viewing HTML documents. ESpec's internal table editor can also be used if desired. In order to changed these settings, press the "Advanced" button in the "ES-Fit settings" window.



1 IGNORED
2 PASSED
3 PASSED
4 PASSED
5 PASSED

Windows HTML viewer: explorer

Unix HTML viewer: netscape

Windows HTML editor: Office\OFFICE11\frontpg.exe"

Unix HTML editor: netscape

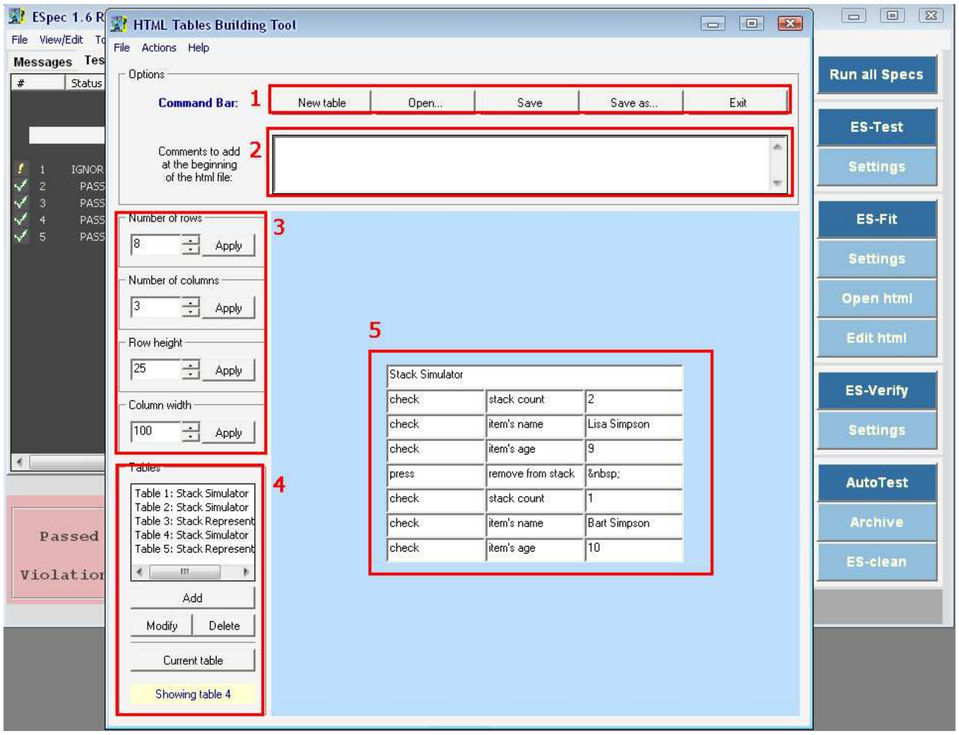
Use EiffelSpec's HTML editor ☑

Default output extension: _out

Passed 4
Violations 0
Total 4

Internal vs. External HTML editor: (1) Windows HTML viewer: this is the user specified external HTML viewing tool for windows (Explorer by default) to be used by ESpec (2) Unix HTML viewer: HTML viewer for Linux users (3) Windows HTML Editor: The HTML editor tool for Windows (4) Linux HTML Editor: The HTML editor tool for Linux (5) If selected, the internal ESpec HTML editor will be used (6) Default output extension: this is the string that will be concatenated to the name of the generated output HTML documents ("_out" by default).

Figure D.19: ES-Fit Advanced options



Number of rows: 8 Apply

Number of columns: 3 Apply

Row height: 25 Apply

Column width: 100 Apply

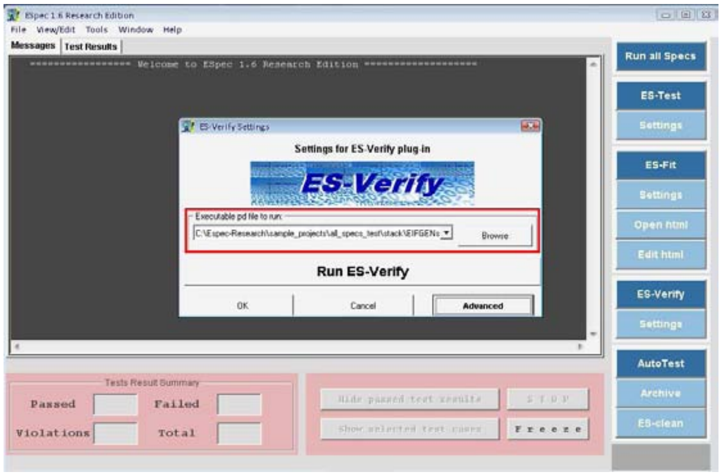
Table 1: Stack Simulator
Table 2: Stack Simulator
Table 3: Stack Represent
Table 4: Stack Simulator
Table 5: Stack Represent

Showing table 4

Stack Simulator		
check	stack count	2
check	item's name	Lisa Simpson
check	item's age	9
press	remove from stack	
check	stack count	1
check	item's name	Bart Simpson
check	item's age	10

ES-Fit internal HTML table editor: ESpec's internal HTML editor allows users to easily view or edit HTML Fit tables. This editor will automatically open when the user has chosen it from the ES-Fit advanced setting window. (1) New Table: creates a new table in the input HTML file, Open: opens an HTML input file containing multiple tables, Save: saves the current HTML document. Save as: allows to save another copy of the file with different name. Exit: exits the editor tool. (2) This text area is used to add comments in front of each table. (3) Number of rows/columns of a selected table can be modified here (width of each row/column can also be modified). (4) All tables in the opened HTML document are listed here. To edit a table, select the title from the list and the table appears in the display area. Any changes to the table have to be saved using the "Modify" button. To add a new table press "Add" and to remove a table select the title from the list and press "Delete". (5) Main display area to edit the tables.

Figure D.20: ESpec internal HTML editor

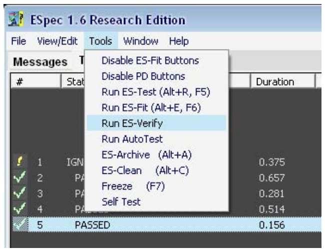


```
================= Welcome to ESpec 1.6 Research Edition ===================
```

Settings for ES-Verify plug-in

Executable pd file to run: C:\Espec-Research\sample_projects\all_specs_test\stack\EIFGENs

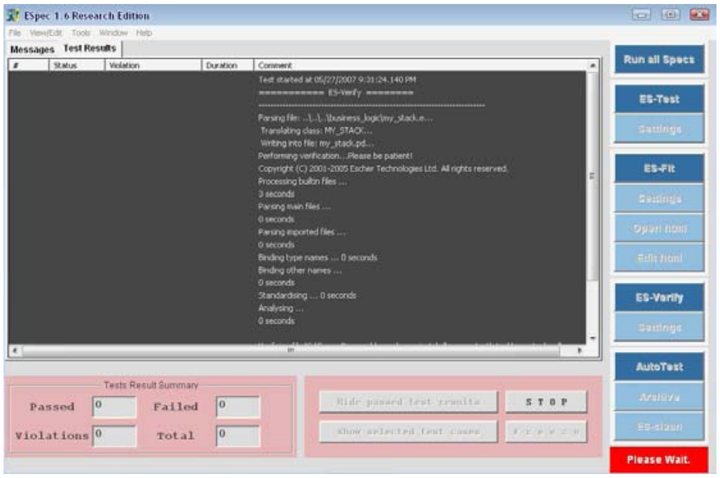
ES-Verify settings: ES-Verify component of ESpec, translates the input Eiffel files (which are specified in the test suite) to the Perfect Developer language and then runs the Perfect Developer theorem prover on the translated files. For settings, press the "Settings" button under "ES-Verify" button in the main window. This opens the "ES-Verify settings" window. User must select the project executable located in EIFGENs folder.



#	Stat	Duration
1	IGN	0.375
2	PA	0.657
3	PA	0.281
4	PA	0.514
5	PASSED	0.156

Running ES-Verify tool To run ES-Verify, select "Run ES-Verify" from the "Tools" menu.

Figure D.21: ES-Verify settings



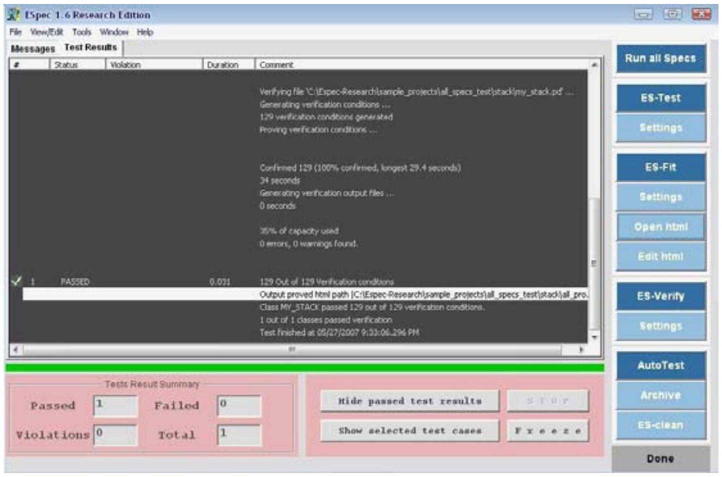
```
#   Status   Violation   Duration   Comment
                                    Test started at 05/27/2007 9:31:24.140 PM
                                    =========== ES-Verify ========
                                    ================================================
                                    Parsing file: ..\..\..\business_logic\my_stack.e...
                                     Translating class: MY_STACK...
                                     Writing into file: my_stack.pd...
                                    Performing verification....Please be patient!
                                    Copyright (C) 2001-2005 Escher Technologies Ltd. All rights reserved.
                                    Processing builtin files ...
                                    3 seconds
                                    Parsing main files ...
                                    0 seconds
                                    Parsing imported files ...
                                    0 seconds
                                    Binding type names ... 0 seconds
                                    Binding other names ...
                                    0 seconds
                                    Standardising ... 0 seconds
                                    Analysing ...
                                    0 seconds
```

Tests Result Summary

Passed 0 Failed 0

Violations 0 Total 0

ES-Verify in progress: ES-Verify tool runs as a separate thread inside ESpec tool. This process is CPU intensive and usually takes a long time. User should wait for the process to finish. A syntax error in the input Eiffel file or a translation error will stop the process automatically and the failure will be reported to the GUI. ES-Verify can also be stopped manually at any time by the user if the "Stop" button is pressed.



```
#   Status   Violation   Duration   Comment
                                    Verifying file 'C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd' ...
                                    Generating verification conditions ...
                                    129 verification conditions generated
                                    Proving verification conditions ...


                                    Confirmed 129 (100% confirmed, longest 29.4 seconds)
                                    34 seconds
                                    Generating verification output files ...
                                    0 seconds

                                    35% of capacity used
                                    0 errors, 0 warnings found.


✓ 1  PASSED              0.031      129 Out of 129 Verification conditions
                                    Output proved html path [C:\Espec-Research\sample_projects\all_specs_test\stack\all_pro.
                                    Class MY_STACK passed 129 out of 129 verification conditions.
                                    1 out of 1 classes passed verification
                                    Test finished at 05/27/2007 9:33:06.296 PM
```

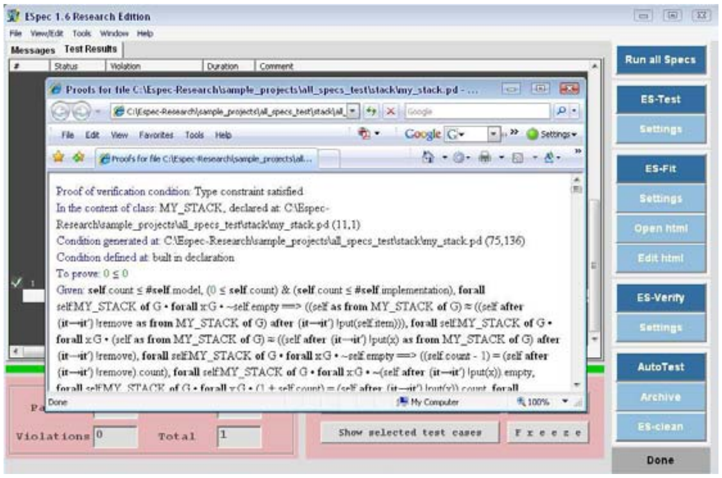
Tests Result Summary

Passed 1 Failed 0

Violations 0 Total 1

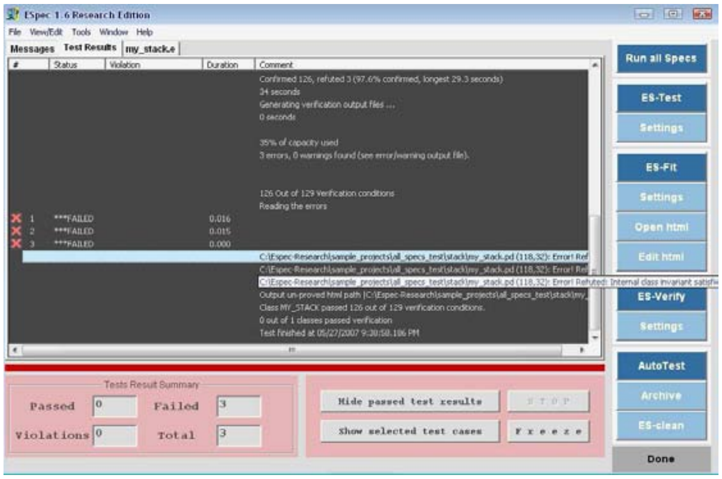
ES-Verify results: The result of running the theorem prover will be displayed inside ESpec main window. If all proof obligations are discharged successfully, the green bar will be shown. User can see the generated HTML proof file by double clicking on it ("output proof html path").

Figure D.22: ES-Verify results (passed)


ESpec 1.6 Research Edition
File View/Edit Tools Window Help
Messages Test Results
# Status Violation Duration Comment
Proofs for file C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd - ...
C:\Espec-Research\sample_projects\all_specs_test\stack\all
Google
File Edit View Favorites Tools Help
Google G Settings
Proofs for file C:\Espec-Research\sample_projects\all...
Done
My Computer
100%
Violations 0
Total 1
Show selected test cases
Freeze
Run all Specs
ES-Test
Settings
ES-Fit
Settings
Open html
Edit html
ES-Verify
Settings
AutoTest
Archive
ES-clean
Done

```
Proof of verification condition: Type constraint satisfied
In the context of class: MY_STACK, declared at: C:\Espec-
Research\sample_projects\all_specs_test\stack\my_stack.pd (11,1)
Condition generated at: C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd (75,136)
Condition defined at: built in declaration
To prove: 0 ≤ 0
Given: self.count ≤ #self.model, (0 ≤ self.count) & (self.count ≤ #self.implementation), forall
self:MY_STACK of G • forall x:G • ~self.empty ==> ((self as from MY_STACK of G) ≈ ((self after
(it—it') !remove as from MY_STACK of G) after (it—it') !put(self.item))), forall self:MY_STACK of G •
forall x:G • (self as from MY_STACK of G) ≈ ((self after (it—it') !put(x) as from MY_STACK of G) after
(it—it') !remove), forall self:MY_STACK of G • forall x:G • ~self.empty ==> ((self.count - 1) = (self after
(it—it') !remove).count), forall self:MY_STACK of G • forall x:G • ~(self after (it—it') !put(x)).empty,
forall self:MY_STACK of G • forall x:G • (1 + self.count) = (self after (it—it') !put(x)).count, forall
```

HTML proof file: When user chooses to see the generated HTML proof file, ESpec opens the generated file in the HTML viewer.

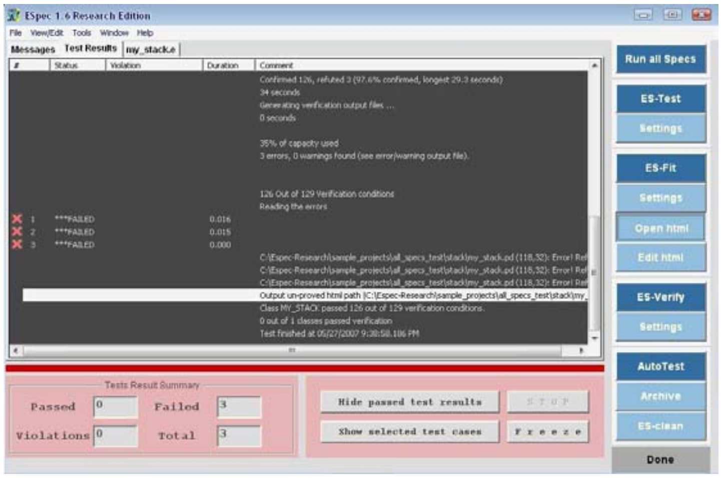


#	Status	Violation	Duration	Comment
				Confirmed 126, refuted 3 (97.6% confirmed, longest 29.3 seconds)
				34 seconds
				Generating verification output files ...
				0 seconds
				35% of capacity used
				3 errors, 0 warnings found (see error/warning output file).
				126 Out of 129 Verification conditions
				Reading the errors
✗ 1	***FAILED		0.016	
✗ 2	***FAILED		0.015	
✗ 3	***FAILED		0.000	
				C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd (118,32): Error! Ref
				C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd (118,32): Error! Ref
				C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd (118,32): Error! Refuted: Internal class invariant satisfie
				Output un-proved html path [C:\Espec-Research\sample_projects\all_specs_test\stack\my_
				Class MY_STACK passed 126 out of 129 verification conditions.
				0 out of 1 classes passed verification
				Test finished at 05/27/2007 9:30:58.186 PM

Passed	0	Failed	3
Violations	0	Total	3

ES-Verify failures: The failures (if any) are reported to the ESpec GUI. These errors can lead the developer to the location of the problem in the original Eiffel file.

Figure D.23: ES-Verify failures

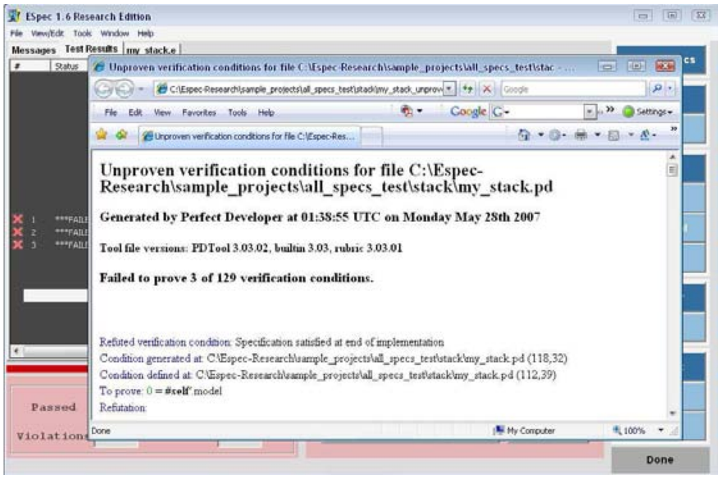


#	Status	Violation	Duration	Comment
				Confirmed 126, refuted 3 (97.6% confirmed, longest 29.3 seconds)
				34 seconds
				Generating verification output files ...
				0 seconds
				35% of capacity used
				3 errors, 0 warnings found (see error/warning output file).
				126 Out of 129 Verification conditions
				Reading the errors
1	***FAILED		0.016	
2	***FAILED		0.015	
3	***FAILED		0.000	
				C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd (118,32): Error! Ref
				C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd (118,32): Error! Ref
				C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd (118,32): Error! Ref
				Output un-proved html path |C:\Espec-Research\sample_projects\all_specs_test\stack\my_
				Class MY_STACK passed 126 out of 129 verification conditions.
				0 out of 1 classes passed verification
				Test finished at 05/27/2007 9:38:58.106 PM

Tests Result Summary

Passed	0	Failed	3
Violations	0	Total	3

Opening the unproven HTML: User can request to see the unproven rules by double clicking on the "Output un-proven html path".



ESpec 1.6 Research Edition
File View/Edit Tools Window Help
Messages Test Results my_stack.e
Unproven verification conditions for file C:\Espec-Research\sample_projects\all_specs_test\stac - ...
C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack_unprov
Google
File Edit View Favorites Tools Help
Google
Settings
Unproven verification conditions for file C:\Espec-Res...
Done
My Computer
100%
Done
Passed
Violations

Unproven verification conditions for file C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd

Generated by Perfect Developer at 01:38:55 UTC on Monday May 28th 2007

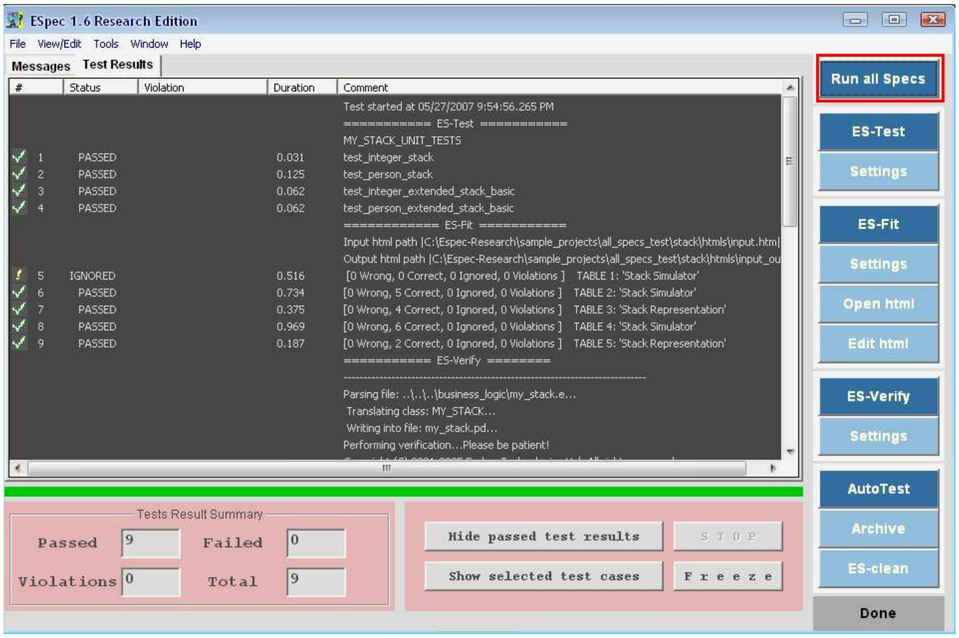
Tool file versions: PDTool 3.03.02, builtin 3.03, rubric 3.03.01

Failed to prove 3 of 129 verification conditions.

Refuted verification condition: Specification satisfied at end of implementation
Condition generated at: C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd (118,32)
Condition defined at: C:\Espec-Research\sample_projects\all_specs_test\stack\my_stack.pd (112,39)
To prove: 0 = #self'.model
Refutation:

Unproven HTML: The un-proven HTML will be opened in a new window. This output will help the expert developer (with PD knowledge) to fix the code.

Figure D.24: ES-Verify failures cont.



```
#     Status     Violation     Duration   Comment
                                          Test started at 05/27/2007 9:54:56.265 PM
                                          =========== ES-Test ===========
                                          MY_STACK_UNIT_TESTS
✓ 1   PASSED                   0.031      test_integer_stack
✓ 2   PASSED                   0.125      test_person_stack
✓ 3   PASSED                   0.062      test_integer_extended_stack_basic
✓ 4   PASSED                   0.062      test_person_extended_stack_basic
                                          ============ ES-Fit ===========
                                          Input html path |C:\Espec-Research\sample_projects\all_specs_test\stack\htmls\input.htm|
                                          Output html path |C:\Espec-Research\sample_projects\all_specs_test\stack\htmls\input_ou
! 5   IGNORED                  0.516       [0 Wrong, 0 Correct, 0 Ignored, 0 Violations ]   TABLE 1: 'Stack Simulator'
✓ 6   PASSED                   0.734      [0 Wrong, 5 Correct, 0 Ignored, 0 Violations ]   TABLE 2: 'Stack Simulator'
✓ 7   PASSED                   0.375      [0 Wrong, 4 Correct, 0 Ignored, 0 Violations ]   TABLE 3: 'Stack Representation'
✓ 8   PASSED                   0.969      [0 Wrong, 6 Correct, 0 Ignored, 0 Violations ]   TABLE 4: 'Stack Simulator'
✓ 9   PASSED                   0.187      [0 Wrong, 2 Correct, 0 Ignored, 0 Violations ]   TABLE 5: 'Stack Representation'
                                          =========== ES-Verify ========
                                          ------------------------------------------------------------------------
                                          Parsing file: ..\..\..\business_logic\my_stack.e...
                                           Translating class: MY_STACK...
                                           Writing into file: my_stack.pd...
                                          Performing verification...Please be patient!
```

Tests Result Summary

Passed	9	Failed	0
Violations	0	Total	9

Running all ESpec tools at the same time: For regression testing, user is encouraged to run all the tests (i.e., ES-Test, ES-Fit and ES-Verify components) after each modification. Of course the system has to be freezed every time there is a change in the system under test. In order to run all the tests, press "Run all specs" on the ESpec GUI. This option invokes ES-Test, ES-Fit and ES-verify and collects their results under a single green/red bar.

Figure D.25: Running all the tests

E Appendix: Misc

E.1 CREDIT_FIXTURE

```
class CREDIT_FIXTURE inherit
  ES_COLUMN_FIXTURE
create
  make

feature {NONE}
  make
    do
      bind ("Should be given credit?", agent allow_credit)
      bind ("Maximum credit allowed", agent credit_limit)
    end

  allow_credit (m: INTEGER; b: REAL): BOOLEAN
      -- m, b are 'months' and 'balance' inputs
    do
      if (m >= 12 and m < 24) and b < 60000.00 then
        Result := true
      elseif (m >= 24) then
        Result := true
      end
    end

  credit_limit (m: INTEGER; b: REAL): REAL
    do
      if (m >= 12 and m < 24) and b < 60000.00 then
        Result := 100000.00
      elseif m > 24 then
        Result := 200000.00
      else
        Result := 0.00
      end
    end

end -- class CREDIT_FIXTURE
```

E.2 NEW_CREDIT_FIXTURE

```
class NEW_CREDIT_FIXTURE inherit
  ES_COLUMN_FIXTURE
  redefine
    process_row,
    post_process_table
  end

create
  make

feature {NONE}
  make
    do
      bind ("Should be given credit?", agent allow_credit)
      bind ("Maximum credit allowed", agent credit_limit)
      bind ("Total Credit", agent credit_sum)
    end

  allow_credit (m: INTEGER; b: REAL): BOOLEAN
      -- m, b are 'months' and 'balance' inputs
    do
      if (m >= 12 and m < 24) and b < 60000.00 then
        Result := true
      elseif (m >= 24) then
        Result := true
      end
    end

  credit_limit (m: INTEGER; b: REAL): REAL
    do
      if (m >= 12 and m < 24) and b < 60000.00 then
        Result := 100000.00
      elseif m > 24 then
        Result := 200000.00
      else
        Result := 0.00
      end

      credit_sum := credit_sum + b -- new code for collecting the credit
    end

  process_row is
      -- redefined: ignores the last row
    do
      if not (content_under_heading ("Should be given credit?").is_equal
        ("Total Credit")) then
          Precursor -- if it is not the last row, process as a row fixture
      end -- if it is the last row, ignore it
    end

  post_process_table is
    do
      connect_to_target ("Total Credit", "Maximum credit allowed")
      execute_cell ("Total Credit")
    end

  credit_sum: REAL -- collects the credit

end -- class NEW_CREDIT_FIXTURE
```

Color mapping chart for black and white copies	
Yellow	🟨
Green	🟩
Red	🟥
Gray	⬜

Table E.1: Color mapping for understanding the black and white copies of this thesis

Bibliography

[1] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, February 2005.

[3] Scott Ambler. Agile Model Driven Development is Good Enough. *IEEE Software*, 20(5):71–73, 2003. Agile Model Driven Development is Good Enough.

[4] Tatiana Andronache. The english language as an effective IT tool. *Computerworld*, page 18, Feb. 2007.

[5] Ralph Johan Back and Joakim von Wright. *Refinement Calculus*. Springer-Verlag, New York, 1998. Refinement Calculus.

[6] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of FMCO*, 2005.

[7] Mike Barnett, Robert DeLine, Manuel Fahndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[8] Mike Barnett, Robert DeLine, Bart Jacobs, Manuel Fhndrich, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# Programming System: Challenges and Directions. *Position paper at VSTTE*, 2005.

[9] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004*. Springer Verlag, LNCS 3362, 2004.

[10] Kent Beck. *Test-driven development: by example*. Addison-Wesley, Boston, 2003.

[11] Kent Beck, Alistair Cockburn, R Ron Jeffries, and J. Highsmith. Agile Manifesto www.agilemanifesto.org/history.html. Technical report, 2001.

[12] Bernhard Beckert and Vladimir Klebanov. A dynamic logic for deductive verification of concurrent programs. In Mike Hinchey and Tiziana Margaria, editors, *Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), London, UK*. IEEE Press, 2007. To appear.

[13] Daniel M. Berry. Formal methods: the very idea — Some thoughts about why they work when they work. *Science of Computer Programming*, 42(1):11–27, 2002. citeseer.nj.nec.com/berry99formal.html.

[14] M. Berry, K. Daudjee, J. Dong, I. Fainchtein, A. Nelson, T. Nelson, and L. Ou. User's manual as a requirements specification: case studies. *Requir. Eng.*, 9(1):67–82, 2004.

[15] D. Bjorner and L. Druffel. Position statement: ICSE-12 workshop on industrial experience using formal methods. In *ICSE '90: Proceedings of the 12th international conference on Software engineering*, pages 264–266, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[16] G. Brat and W. Visser. Combining static analysis and model checking for software analysis. IEEE, 2001. Combining static analysis and model checking for software analysis.

[17] Achim D. Brucker and Burkhart Wolff. A Proposal for a Formal OCL Semantics in Isabelle/Hol. In *Theorem Proving in Higher Order Logics*, volume LNCS 2410. Springer-Verlag, 2002.

[18] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, 2003.

[19] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[20] Gareth Carter, Rosemary Monahan, and Joseph M. Morris. Software refinement with perfect developer. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 363–373, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/-Java2. In Springer-Verlag, editor, *Formal Methods for Components and Objects (FMCO'2005)*, LNCS, 2006.

[22] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[23] Yoonsik Cheon, Leavens, and Gary T. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. (01–12), 2001.

[24] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: cleanly supporting abstraction in design by contract. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[25] Leitner A. Ciupa, I. Automatic testing based on design by contract. In *Proceedings of Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World)*, pages 545–557, September 19-22 2005.

[26] Tony Clark, Jos B. Warmer, and Lecture Notes in computer science York University. *Object modeling with the OCL : the rationale behind the Object Constraint Language*. Springer-Verlag Berlin Heidelberg, Berlin ; New York, 2002.

[27] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[28] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2. Technical Report NIII-R0413, Nijmegen Institute for Computing and Information Science, May 2004.

[29] David Crocker. Perfect Developer: A tool for Object-Oriented Formal Specification and Refinement. In *Tools Exhibition Notes at Formal Methods Europe*, 2003.

[30] David Crocker. Safe Object-Oriented Software: The Verified Desing-By-Contract Paradigm. In F.Redmill & T.Anderson, editor, *Twelfth Safety-Critical Systems Symposium*, pages 19–41. Springer-Verlag, London, 2004.

[31] Ward Cunningham. Framework for Integrated Test, Java Platform, http://fit.c2.com/wiki.cgi?JavaPlatform, 2005.

[32] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[33] Alan Dix, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-computer interaction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[34] ECMA. Eiffel: Analysis, design and programming language. Standard ECMA-367 (2nd edition), June 2006.

[35] Escher Technologies. *Perfect Developer Language Reference Manual*, 3.0 edition, December 2004. Available from www.eschertech.com.

[36] Ingo Feinerer. Formal Program Verification: a Comparison of Selected Tools and Their Theoretical Foundations. Master's thesis, Vienna University of Technology, January 2005.

[37] Ingo Feinerer and Gernot Salzer. Automated tools for teaching formal software verification. In P. Boca, J.P. Bowen, and D.A. Duce, editors, *Proceedings of Teaching Formal Methods: Practice and Experience*, Electronic Workshops in Computing (eWiC), pages 15–19, BCS London Office, UK, 15 December 2006. British Computing Society, BCS.

[38] Martin Fowler and Kendall Scott. *UML distilled : applying the standard object modeling language*. Addison Wesley Longman, Reading, Mass., 1997.

[39] Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Softw*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[40] Jr. Frederick P. Brooks. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[41] Erich Gamma and K. Kent Beck. JUnit: A cook's tour, 1999.

[42] Marie-Claude Gaudel. Formal specification techniques (extended abstract). In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 223–227, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[43] Robert L. Glass. *Software Engineering: Facts and Fallacies*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[44] Robert L. Glass. The standish report: does it really describe a software crisis? *Commun. ACM*, 49(8):15–16, 2006.

[45] Standish Group. Project management: The criteria for success. Software Magazine, February 2001.

[46] Sam Guckenheimer and Juan J. Perez. *Software Engineering with Microsoft Visual Studio Team System (Microsoft .NET Development Series)*. Addison-Wesley Professional, 2006.

[47] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993. Larch: Languages and Tools for Formal Specification.

[48] Bill Hetzel. *The complete guide to software testing (2nd ed.)*. QED Information Sciences, Inc., Wellesley, MA, USA, 1988.

[49] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[50] Engelbert Hubbers. Integrating tools for automatic program verification. In *Ershov Memorial Conference*, pages 214–221, 2003.

[51] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. SpringerVerlag, 2005.

[52] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in C# with NUnit*. The Pragmatic Programmers, 2004.

[53] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[54] Michael Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[55] Michael Jackson. *Problem frames: analyzing and structuring software development problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[56] Bart Jacobs and Erik Poll. Java program verification at nijmegen: Developments and perspective. In *ISSS*, pages 134–153, 2003.

[57] Loryn Jenkins. Eiffel to C++ terminology mapping.

[58] C.B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1986. Systematic Software Development using VDM.

[59] Greg Nelson K. Rustan M. Leino and James B. Saxe. *ESC/Java User's Manual*, 2000. http://research.compaq.com/SRC/esc/papers.htm.

[60] Gary T. Leavens, K. Rustan M. Leino, and Peter Muller. Specification and verification challenges for sequential object-oriented programs. TR 06-14, Department of Computer Science, Iowa State University, May 2006.

[61] Andreas Leitner, Patrick Eugster, Manuel Oriol, and Ilinca Ciupa. Reflecting on an existing programming language. In *Proceedings of TOOLS EUROPE 2007 - Objects, Models, Components, Patterns,*, July 2007.

[62] R. C. Martin. The Liskov substitution principle. 8(3):14, 16–17, 20–23, March 1996.

[63] Robert C. Martin. *UML for Java Programmers*. Prentice Hall, 2003. UML for Java Programmers.

[64] Bertrand Meyer. Design by Contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[65] Bertrand Meyer. Applying ”Design by Contract”. *Computer (IEEE)*, 25:40–51, 1992.

[66] Bertrand Meyer. *Eiffel the Language*. Prentice Hall, 1992. Eiffel the Language.

[67] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[68] Rick Mugridge and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests*. Prentice-Hall, 2005.

[69] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. pages 35–46, Limerick, Ireland, 2000.

[70] OMG. OMG Unified Modeling Language Specification: Version 1.4. Technical report, 2001. OMG Unified Modeling Language Specification: Version 1.4.

[71] Jonathan Ostroff, Faraz Torshizi, and Hai Feng Huang. Verifying properties beyond contracts of SCOOP programs. In *First International Symposium on Concurrency, Real-Time and Distribution in Eiffel-like Languages (CORDIE'06)*, 2006.

[72] Jonathan Ostroff, Chen-Wei Wang, Eric Kerfoot, and Faraz Ahmadi Torshizi. Automated model-based verification of object oriented code. In *Verified Software: Theories, Tools, Experiments (VSTTE Workshop, Floc 2006)*. Microsoft Research MSR-TR-2006-117, 2006.

[73] Jonathan S. Ostroff, Richard F. Paige, David Makalsky, and Phillip J. Brooke. E-tester: a contract-aware and agent-based unit testing framework for eiffel. *Journal of Object Technology*, 4(7):97–114, 2005.

[74] Jonathan S. Ostroff and Faraz Ahmadi Torshizi. Testable Requirements and Specifications. In Bertrand Meyer and Yuri Gurevich, editors, *Tests and Proofs (TAP'07)*, volume LNCS 4454. Springer Verlag, 2007.

[75] Richard Paige and Jonathan S. Ostroff. From Z to Bon/Eiffel. Hawaii, 1998. IEEE Computer Society. www.cs.yorku.ca/techreports/1998/CS-98-05.html.

[76] Richard Paige and Jonathan S. Ostroff. The Single Model Principle. *Journal of Object Oriented Technology*, 1(5), 2002.

[77] Richard F. Paige and Jonathan S. Ostroff. Developing BON as an Industrial-Strength Formal Method. volume LNCS 1708. Springer-Verlag, 1999. Developing BON as an Industrial-Strength Formal Method.

[78] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[79] David Lorge Parnas and Paul C. Clements. A Rational Design Process: How and Why to Fake it. *IEEE Trans. on Software Engineering*, SE-12(2):251–257, 1986. A Rational Design Process: How and Why to Fake it.

[80] William N. Robinson, Suzanne D. Pawlowski, and Vecheslav Volkov. Requirements interaction management. *ACM Comput. Surv.*, 35(2):132–190, 2003.

[81] Robert Seater and Daniel Jackson. Problem frame transformations: deriving specifications from requirements. In *IWAAPF '06: Proceedings of the 2006 international workshop on Advances and applications of problem frames*, pages 71–80, New York, NY, USA, 2006. ACM Press.

[82] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003. The Pragmatics of Model-Driven Development.

[83] J.M. Spivey. *The Z Notation: A Reference Manual (2nd edition)*. Prentice-Hall, Englewood Cliffs, N.J., 1992.

[84] Friedrich Steimann. The paradoxical success of aspect-oriented programming. *SIGPLAN Not.*, 41(10):481–497, 2006.

[85] Brian Stevens. Implementing Object-Z with PerfectDeveloper. *Journal of Object Technology*, 6(2):189–202, March-April 2006.

[86] Inc. The Standish Group International. Extreme chaos. Technical report, 2001.

[87] Kim Walden and Jean-Marc Nerson. *Seamless Object Oriented Software and Architecture*. Prentice Hall, 1995. Seamless Object Oriented Software and Architecture.

[88] Jos Warmer and Anneke Kleppe. *The Object Constraint Language Second Edition, Getting Your Models Ready for MDA*. Addison-Wesley, Boston, 2003.

[89] Karl Wiegers. Writing Quality Requirements. *Softw. Dev.*, 7(5):44–48, 1999.

[90] Eric S. K. Yu. Models for supporting the redesign of organizational work. In *COCS '95: Proceedings of conference on Organizational computing systems*, pages 226–236, New York, NY, USA, 1995. ACM Press.