ESpec – a Tool for Agile Development via Early Testable Specifications

Faraz Ahmadi Torshizi and Jonathan S. Ostroff

Technical Report CS-2006-04

May 24, 2006

Department of Computer Science and Engineering

4700 Keele Street North York, Ontario M3J 1P3 Canada

ESpec – a Tool for Agile Development via Early Testable Specifications

Faraz Ahmadi Torshizi
Department of Computer Science and Engineering, York University, Canada
faraz@cs.yorku.ca

Jonathan S. Ostroff
Department of Computer Science and Engineering, York University, Canada
jonathan@cs.yorku.ca

Abstract

The ESpec (Eiffel Specification Tool) is a unified environment that allows software developers to combine Fit tables (for customer acceptance tests) with Early Testable Specifications (Contracts and Unit Tests). This means that a single integrated tool can be used to develop and test the requirements, design and implementation of a software product. Since Fit tables, contracts and Unit Tests accumulate, we get regression testing every time the tool is invoked. The regression testing makes it easier to refactor a design, as all the properties are checked for every design change. ESpec includes a fully automated formal verification tool that uses a theorem prover to demonstrate that the code satisfies its specification. These tools, either individually or in concert, allow the developer to certify the quality of the product in a variety of ways.

1. Introduction

It is generally recognized that the production of a quality software product requires suitable Validation and Verification [1]. Informally, in Validation we ask the question: Am I building the right product? In Verification we ask the question: Am I building the product right? The combination of Validation and Verification allows us to certify the quality of the product.

There are many qualities of interest such as efficiency, portability and understandability. The quality of interest in this paper is software reliability. A product is reliable if it is correct (performs its tasks according to specification) and robust (reacts appropriately to abnormal conditions).

This paper builds on the contributions of agile methods, especially the Fit framework for customer acceptance tests [12] and Test Driven Development (TDD) via Unit Testing [3].

Bad software is often due to misunderstood requirements, vague specifications, and late development of the code. By contrast, TDD and the Fit framework are built on the recognition that automated tests are "relentlessly concrete" [12]. Requirements must be clarified and precisely expressed (in the form of a Test) early in the development process in such way that they can be automatically checked right from the start and throughout the process. Early clarification of the requirements (in a Test) also drives the development process and provides feedback when the goals have not been reached.

In this paper, we take these agile methods one step further. We add a lightweight formal method component, combining classical ideas of formal specification and verification with these new automatic checking technologies. This is done by adopting Design by Contract [10]. Contracts are used to express specifications, and contract violations are signaled in both Fit Tables (for Validation) and Unit Tests (for Verification). We also discuss a formal integrated verification module using a fully automated theorem prover (called Perfect Developer).

We illustrate the method using the Eiffel language which has a mature contracting mechanism, but the conceptual ideas could be used in any of the emerging contracting languages such as Spec# [11] and ESC/Java [14]. To illustrate the concepts we provide a tool called *ESpec*[1] that allows the developer to combine a variety of tests in one test suite. ESpec consists of *ES-Fit* for Validation, *ES-Test* (for lightweight Verification) and *ES-Perfect* (for full Verification). The checks for Validation and Verification can be run on their own or together with a unified report under a single green/red bar. If all the checks succeed, then we have certified the quality to the level asserted by the various tests and specifications.

[1] ESpec website: www.cs.yorku.ca/~sel/espec. This is the first implementation of Fit in Eiffel.

We have also extended the Fit framework to add more flexibility to customer acceptance tests. We illustrate the concepts and the tool with a small example, chosen to describe a complete cycle of development from informal requirements to certified code. However, the tool has been used on larger problems such as testing the code of students developing games and game frameworks. This paper will also act as an overview of ESpec tool.

2. Requirements and Specifications

In this section we provide a brief formal schema for thinking about Requirements and Specifications, where these terms have the meaning described in [7, 8, 13]. The phenomena of the real-world (W) determine the customer's *Requirements* (R). For example, to evaluate if a client pays reliably, we may need to examine phenomena such as invoices and delinquency charges in W. Requirements are normally stated informally or expressed as Stories or Use Cases, and should refer only to a subset of the phenomena in W.

The final software code (or machine M) is all about the phenomena of the program (e.g. data structures such as linked lists, arrays or binary trees). A Specification (S) is a bridge between the phenomena of the real world (W) and the phenomena of the machine (M), describing phenomena (inputs and outputs) at the interface or intersection of W and M. A rational software development proceeds as follows:

- Elicit and document the Requirements R of the customer in terms of the phenomena of W.
- From the Requirements, derive a Specification S for the software code that must be developed.
- From the Specification, derive a machine M (the code).

To certify that the code meets its requirements, we must formally reason as follows:

1. **Validation**: $W \wedge S \Rightarrow R$
2. **Verification**: $M \Rightarrow S$
3. **Certification:** From (1) and (2) conclude that $W \wedge M \Rightarrow R$

In our framework, Validation is performed by checking Fit Tables (ES-Fit), lightweight Verification is performed by running Unit Tests (ES-Test), and full Verification is performed by ES-Perfect. Thus, Fit Tables are our customer Requirements and Contracts are the Specification of the machine. Since Specification S occurs in both (1) and (2), any violation of the contract is reported both in the Fit Tables and Unit Tests. If all the checks succeed, then we can claim that we have certified the product to the level specified by the tests. Our formal schema provides methodological guidance (from informal requirements, through to the design and coding) for using the toolset which satisfies the following desirable properties. The toolset is fully automatic requiring only that the user take the time to write *early testable* specifications and requirements – *early* because they can be written up-front, long before the code is implemented and *testable* because they can be checked automatically. Early testable requirements and specifications allow us to transform informal goals into concrete and precise descriptions required for Validation and Verification. The toolset can be used on real software of any size and the green/red bar measures our progress towards meeting the certification criteria.

3. Credit example

Consider the following business rule [12]:

> [R1] *Credit is allowed, up to an amount of $1,000, for a customer who has been trading with us for more than 12 months, has paid reliably over that period, and has a balance owing of less than $6,000.*

Calculate Credit 1				
Months	*Reliable*	*Balance*	Allow Credit*	Credit Limit*
14	true	5000	true	1000
13	true	3000	true	1000
24	false	0	false	0
18	true	6000	false	0
12	true	5500	true	1000

Figure 1. HTML Fit Table for requirement [R1]

Our client may create the HTML Fit table shown in Figure 1 as a concrete and testable description of the business rule without needing to know too much about programming. The acceptance tests in Figure 1 may be written in any HTML editor. 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 three headings (*Months*, *Reliable*, and *Balance*) are the given inputs. The next two headings (*Allow Credit* and *Credit Limit*) are the calculated values for testing creditworthiness of a customer. The "*" or any other string may be used to indicate that these are outputs to be calculated based on the input values. Each row of the table describes such a calculation. In this case, each row is an independent test case, e.g. the third row illustrates a correct credit calculation, i.e., the customer has been trading for more than 12

months and has paid reliably over that period with a balance less than $6,000.

The client should be able to run the table against the code that developers have been working on, and see the output shown in Figure 2.

Calculate Credit 1				
Months	*Reliable*	*Balance*	Allow Credit*	Credit Limit*
14	true	5000	true	1000
13	true	3000	true	1000
24	false	0	false	0
18	true	6000	false	0
12	true	5500	true *Expected* False *Actual*	1000 *Expected* 0 *Actual*

Figure 2. Validating Requirements in Figure 1

The Fit framework provides the software developer with the infrastructure to connect these tables to the software under development via glue code called *Fixtures* which we will discuss in more detail in the sequel. ESpec's Fit engine (ES-Fit) extracts the information from the table, runs the Fixtures which execute the tests defined in the table, and reports any errors back to the table, allowing the customer to browse the results (as shown in Figure 2). The tests in the table show why the approach is helpful:

- The Fit tests help our client (and software developer) to think about and communicate the business needs with concrete examples. The tests transform the informal requirements (e.g. [R1]) into precise testable descriptions.
- By continually and automatically running the Fit tests, the client gains confidence that the code is doing what is expected from the business perspective, and that the requirements continue to be satisfied as the code grows in functionality.

Figure 3 shows a snapshot of the ESpec tool after running some tests. A red bar (as well as a test summary) indicates that there exists at least one test that did not pass. The software developer may either browse or edit the input files or browse the output files to obtain feedback about any errors.

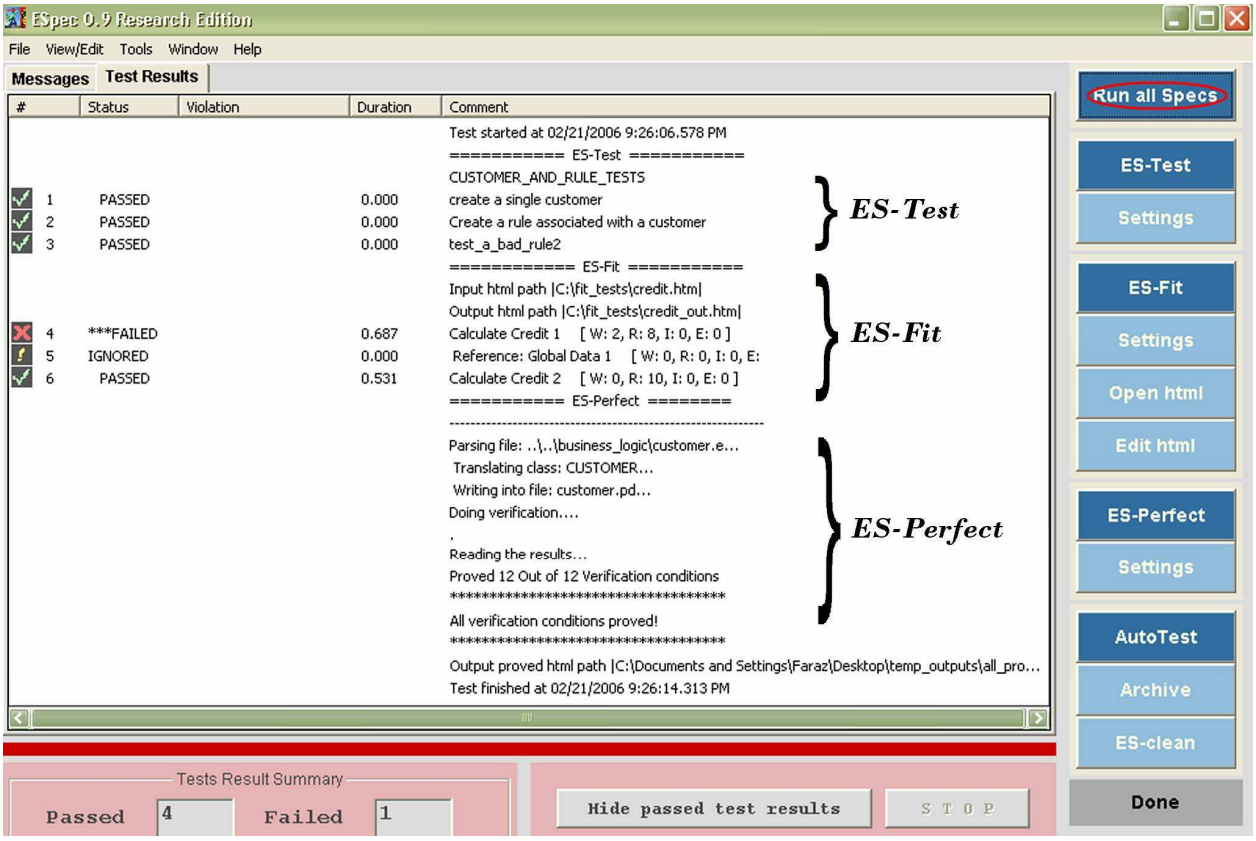


Figure 3. snapshot of ESpec tool

4. Fixture code [ES-Fit]

As mentioned before, the developer needs to provide the Fixture code to connect HTML tables to the underlying code.

```
class CREDIT_FIXTURE inherit ES_COLUMN_FIXTURE create
  make
feature {NONE}
  make is
    do
      bind ("Allow Credit*", agent allow_credit)
      bind ("Credit Limit*", agent credit_limit)
    end

  allow_credit(m:INTEGER;r:BOOLEAN;b:REAL): TUPLE is
      -- `m', `r' and `b' are 'months', 'reliable' and 'balance' inputs
    do
      Result := [m > 12 and r and b < 6000.00]
    end

  credit_limit(m:INTEGER;r:BOOLEAN;b:REAL): TUPLE is
    do
      if m > 12 and r and b < 6000.00 then
        Result := [1000.00]
      else
        Result := [0.00]
      end
    end
  end
```

Figure 4. Fixture code for the table in Figure 1 written in Eiffel

Figure 4 contains the code that a software developer must write to run ES-Fit on *credit.htm* (Figure 1), which will produce the output *credit_out.htm* (Figure 2). For each row of *credit.htm*, we must perform two calculations (*Allow Credit* is performed by routine *allow_credit* and *Credit Limit* by routine *credit_limit*). The *make* creation routine binds the customer-provided name of the calculation in the table to the appropriate routine. This style of Fit test is called a *Column Fixture* and thus class CREDIT_FIXTURE inherits from ES_COLUMN_FIXTURE (provided to the developer by the framework).

ES-Fit runs all the standard Fit tables as specified by the Fit framework. The table in Figure 1, however, is not standard because the calculation headings (*Allow Credit* and *Credit Limit*) are not actual routine or method names. Other Fit tools such as the original Java reference tool [6], use reflection to bind the calculation names to the actual methods. Therefore, these tools require that the customer refer to routine names as they appear in the code, e.g. *allowCredit()*, in the table header. Figure 5 shows the same Fixture code written in Java.

Eiffel does not yet have full internal reflection capabilities (Although there are external libraries that implement reflection for Eiffel [9]). The ES-Fit tool requires the bind instructions in the *make* constructor (Figure 4).

```
public class CalculateCredit1 extends ColumnFixture {

  // Inputs are captured as global variables.
  // Input variables must be declared as public
  // Names of the variables must be exactly the same as
  // they appear on the table headings
  public int months;
  public boolean reliable;
  public double balance;

  // Method names must be exactly the same as appeared on the table
  public boolean allowCredit() {
    return (months > 12 && reliable && balance < 6000.00);
  }

  public double creditLimits() {
    if (months > 12 && reliable && balance < 6000.00)
    {  return 1000.00;
    }
    else
    {  return 0.00;
    }
  }
}
```

Figure 5. Java Fixture code for the table in Figure 1

An advantage of the need to manually bind the calculations is that our clients have the freedom to write whatever calculation headers they prefer without being restricted. Also, the ES-Fit routines may be declared as private (exported to NONE) as opposed to the Java routines which must be public for reflection to work.

Another advantage of the ES-Fit solution is that the input values can be arguments to the calculation routines thus keeping inputs and outputs organized in a single method. In the Java solution, the methods may not take arguments and thus the inputs are captured as public attributes.

For simple projects, the code in Figure 4 suffices; the business logic resides in the Fixture code, e.g. the calculation to allow credit given by:

```
Result := [months > 12 and reliable and balance < 6000]
```

is contained in the Fixture code. Obviously, as the code increases in complexity the developer will want to develop design classes such as RULE and CUSTOMER shown in the business logic package in Figure 11 (The BON notation is similar to UML, except that associations are drawn with a double arrow and inheritance relationships with a single arrow). The job of the Fixture code will be to call the appropriate features of the business logic.

4.1. Reference tables

Consider the following more abstract restatement of the requirements:

> [R2] *Credit is allowed, up to an amount of $X, for a customer who has been trading with us for more than Y months, has paid reliably over that period, and has a balance owing of less than $Z.*

Reference: Global Data 1	
Credit Limit (X)	1000.00
Trading Months (Y)	12
Balance Owing (Z)	6000.00

Figure 6. Reference table for requirement [R2]

The values X, Y and Z could be described via extra columns in Figure 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 Figure 1. Clients may easily change the reference table to test that the code is working correctly. The standard Fit framework does not accommodate such references. We have thus extended the ES-Fit with a **Reference** keyword (Figure 6). Fixture code (e.g. Figure 10) for the table in Figure 1 may refer to the reference table (Figure 6) for the global data. The following code is used to refer to the balance owing constant in the reference table:

```
get_reference ("Global Data 1", <<"Balance Owing (Z)", "?">>).to_real
```

The above expression searches a reference table with heading *Global Data 1* for a row that starts with *Balance Owing (Z)*. The question mark ("?") represents the value in the associated table cell that we wish to retrieve (i.e. $6000). The code for testing [R2] is provided in Figure 10.

5. Test Specifications [ES-Test]

Unit Tests and contracts provide us with the ability to write early, testable specifications. It is obvious that contracts are specifications that describe what the module must do (rather than how to do it). However, a well-written Unit Test can also be a Test Specification. Consider, for example, a routine *root* that calculates the roots of a quadratic equation. A Unit Test (that is not a Test Specification) is shown in Figure 7.

```
test_root_calculation: BOOLEAN is
  local
    m: MATH
    a, b, c: REAL
  do
    a := 1; b := 3; c := 2;
    create m.make (a, b, c)
    Result := m.root = -1 or m.root = -2
  end
```

Figure 7. A Unit Test (not a Test Specification)

By contrast, a Test Specification would define what it means to be a root (Figure 8). The Unit Test (Figure 7) required that we know in advance what the root is (either -1 or -2).

```
test_specification_for_root: BOOLEAN is
  local
    m: MATH
    a, b, c, x: REAL
  do
    a := 1; b := 3; c := 2
    create m.make (a, b, c)
    x := m.root
    Result := (a*x^2 + b*x + c) = 0
  end
```

Figure 8. A Test Specification

An advantage of the Test Specification (Figure 8) is that we can randomly generate parameters *a*, *b* and *c* and allow the definition of a quadratic root to do the work of testing if the roots are correctly calculated. A Test Specification very quickly leads to the correct contracts for the *root* routine as shown in Figure 9.

```
a, b, c: REAL

root: REAL is
    -- root of a quadratic equation
  require
    b^2 - 4*a*c >= 0
  do
    -- implementation...
  ensure
    (a*Result^2 + b*Result + c) = 0
  end
```

Figure 9. Contracts for the root feature

```
class CREDIT_FIXTURE2 inherit ES_COLUMN_FIXTURE create
    make
feature {NONE}
  make is
    do
      bind ("Allow Credit*", agent allow_credit)
      bind ("Credit Limit*", agent credit_limit)
    end

  allow_credit(m:INTEGER;r:BOOLEAN;b:REAL):TUPLE is
    do
      Result := [m > trading_months and r and b < balance_owing]
    end

  credit_limit(m:INTEGER;r:BOOLEAN;b:REAL):TUPLE is
    do
      if m > trading_months and r and b < balance_owing then
        Result := [credit_allowed]
      else
        Result := [0.00]
      end
    end

  credit_allowed: REAL is
  do
    Result := get_reference ("Global Data 1",
                             <<"Credit Limit (X)", "?">>).to_real
  end

  trading_months: INTEGER is
  do
    Result := get_reference ("Global Data 1",
                             <<"Trading Months (Y)", "?">>).to_integer
  end

  balance_owing: REAL is
  do
    Result :=  get_reference ("Global Data 1",
                             <<"Balance Owing (Z)", "?">>).to_real
  end
end
```

Figure 10. Fixture code for requirement [R2] which refers to a reference table

A Test Specification may also be used to describe collaboration between a collection of objects (as will be shown later).

Whether specifications are contracts or tests, they can both be written early, long before the code itself. Furthermore, these specifications are testable. Test Specifications will fail if (a) the collaboration between the various elements fails to produce the anticipated result, or (b) the contracts fail while executing the tests. There is thus a synergy between the contracts and specification tests – together, they are used to check the correctness of our software products.

We may now write a Test Specification that describes the collaboration between these classes in the business logic (Figure 11). A test for a credit worthy customer is shown in Figure 12.

There are some points to notice about the code in Figure 12 that make this Unit Test a Test Specification describing the collaboration of classes CUSTOMER and RULE:

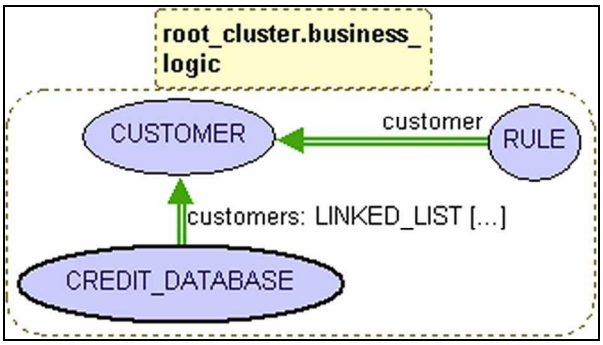


Figure 11. BON diagram of the business logic

```
test_a_good_customer_with_a_rule: BOOLEAN is
  local
    bob: CUSTOMER
    a_rule: RULE
    min_months: INTEGER
    max_balance, max_credit: REAL
    allow_credit: BOOLEAN
  do
    create bob.make ("Bob", 5999, true, 13) – a creditworthy customer
    Result := bob.balance = 5999
              and bob.is_reliable
              and bob.months_trading = 13
    check Result end
    -- setup rule parameters
    min_months := 12; max_balance := 6000.00
    max_credit := 1000.00
    -- [R2]
    allow_credit := bob.balance < max_balance
                    and bob.months_trading > min_months
                    and bob.is_reliable
    -- create a rule
    create a_rule.make (bob, min_months, max_balance, max_credit)
    Result := a_rule.allow_credit = allow_credit
              and a_rule.credit_limit = max_credit
  end
```

Figure 12. Test Specification for a credit worthy customer

- The informal requirement [R2] has been precisely formalized by a Boolean expression *allow_credit*.
- The *allow_credit* formalization will ultimately become a contract in the business logic thus making it part of the delivered code.
- The test describes a collaboration of the two classes CUSTOMER and RULE as shown in the BON dynamic diagram (Figure 13).
- The collaboration (via the Test Specification) is checked by ES-Test.
- The test also provides the names of the relevant features so that it specifies the interfaces of CUSTOMER and RULE. A chart view of class RULE is shown in Figure 14.

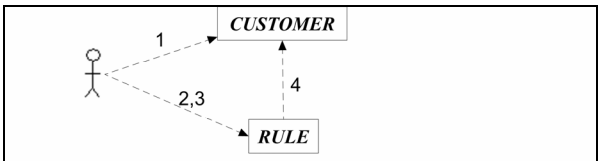
5.1. Contracts

Eiffel has a mature contracting mechanism called Design by Contract. Contracts are written between the user ("client") of a module and the developer ("supplier"). Clients may invoke the module if the precondition is satisfied and the supplier must guarantee the postcondition. Classes are also supplied with class invariants that capture the business rules [10].

A customer may want to check that it does not make sense to apply the business rule [R2] in a case where the months traded is negative. Thus we would expect to see an error in the calculated amount for the first row in Figure 15. We can express that expectation by using the special **Error** keyword. The calculated values will display as green when an error is detected. By declaring the expected value of a cell to be "error", a customer asserts that it is expected that an exception will be generated when running the code.

ES-Fit has the advantage of contracts over the standard Fit frameworks. Contract violations are reported in the Fit tables where appropriate (Figure 15). Is it a precondition violation? Then the calling class is at fault. Is it a postcondition violation? Then the supplier is at fault.

These types of errors are illustrated in Figure 15. In the second row, a precondition violation is reported due to the fact that the balance is negative. In the forth row a postcondition violation is reported. This is due to an implementation error in routine *allow_credit* of class RULE (Figure 16). According to requirement [R2], credit is allowed if a customer has a balance owing *less* than the maximum allowed (in our case $6000).



Scenario: Creating a customer and checking if it satisfies [R2]

1- Create a reliable customer "Bob" with balance $5999 who has been trading for 13 months.
2- Create a rule to check [R2] for "Bob".
3- Check that "Bob" satisfies [R2].
4- allow_credit for "Bob" for the amount of $1000.

Figure 13. Collaboration diagram in BON notation

```
class
  RULE
General
  create: make
Queries
  allow_credit: BOOLEAN
  customer: CUSTOMER
  credit_limit: REAL
  max_balance_owing: REAL
  max_credit_allowed: REAL
  min_months_trading: INTEGER
Commands
  make (a_customer: CUSTOMER;
        min_months: INTEGER;
        max_balance, max_credit: REAL)
```

Figure 14. Chart view of RULE class

Calculate Credit 3			
Months	*Reliable*	*Balance*	Allow Credit*
-1	true	3000	Error
13	true	-1	true **Precondition violated.** **CUSTOMER make @4 r1:** **<000000000176E210> Precondition violated. Fail** **CREDIT_FIXTURE3 allow_credit @1** **<000000000226FBB0> Routine failure. Fail**
24	false	0	false
18	true	6000	false **Postcondition violated.** **RULE allow_credit @3 e1:** **<0000000001B14938> Postcondition violated. Fail** **RULE allow_credit @4** **<0000000001B14938> Routine failure. Fail** **CREDIT_FIXTURE3 allow_credit @3** **<000000000226FBB0> Routine failure. Fail**
12	true	5500	false

Figure 15. Contract violations in Fit tables

```
allow_credit: BOOLEAN is
  do
    Result := customer.balance <= max_balance_owing -- bug
    Result := Result and customer.is_reliable
    Result := Result and customer.months_trading >
                                              min_months_trading
  ensure
    e1:  Result = (customer.balance < max_balance_owing
           and customer.is_reliable
           and customer.months_trading > min_months_trading)
  end
```

Figure 16. Implementation bug in *allow_credit* of class RULE

5.2. Unit Tests

In addition to the Fit acceptance tests, the software developer will also want to write Unit Tests (and Test Specifications) for the CUSTOMER and RULE classes. The contracts in these classes may be seen as test amplifiers, i.e. as the Unit Tests are run, the contracts are also checked and any contract violations are reported. The ESpec Unit Testing engine (ES-Test) provides two types of test cases:

- **Boolean test case:** returns a Boolean value as the result of a test on the system under development. This test case passes if and only if the Boolean result returned is *true* and no contract violation is generated throughout the execution of the test. Details about the contract violations are reported to help in the debugging of the error. We already encountered a Boolean test case in the Test Specification of Figure 12.
- **Violation test case:** this type of test case passes only if an (expected) violation happens while executing this test case. If the test executes without any violations, this type of test case will fail. There are two types of violation cases: the standard case, and the case in which a specific contract violation is expected. If a standard violation is declared, any violation will suffice.

We may use the credit example to illustrate the different kinds of tests. Consider the Boolean test case in Figure 17 in which a rule is created with data that violates the precondition *r1* which specifies that the months traded must not be negative (Figure 19). ES-Test (correctly) reports a precondition violation (tag *r1*) in the *make* routine of class RULE (Figure 18), thus indicating that this Boolean Test fails.

```
test_a_bad_rule1: BOOLEAN is
  local
    bob: CUSTOMER
    a_rule: RULE
  do
    create bob.make ("Bob", 5999, true, 12)
    create a_rule.make (bob, -1, 6000, 1000)
    Result := a_rule.min_months_trading = -1
  end
```

Figure 17. Failing Boolean test for a bad rule

```
test_a_bad_rule1
-------------------------------------------------------------------------
Class / Object    Routine          Nature of exception      Effect
-------------------------------------------------------------------------
RULE              make @4          r1:
<00000000016442F0>                 Precondition violated.   Fail
-------------------------------------------------------------------------
```

Figure 18. Violation is reported by ES-Test

```
make (a_customer: CUSTOMER; min_months: INTEGER; max_balance,
        max_credit: REAL) is
  require
    r1: min_months>=0 and max_credit>=0.0 and max_balance>=0.0
    r2: a_customer /= Void
  ensure
    e1: c = a_customer and min_months_trading = min_months
    e2: max_credit_allowed = max_credit and
        max_balance_owing = max_balance
  end
```

Figure 19. Contract view of the make routine of class RULE

In the test in Figure 17, the precondition violation is expected. Although the Boolean test failed, we would actually like to convert this test into a success. We can do this by converting the Boolean test case *test_a_bad_rule1* (Figure 17) into a Violation test case *test_a_bad_rule2* (Figure 20). The Boolean test is a function routine whereas the Violation test is a command routine that succeeds precisely when the precondition with tag *r1* is violated anywhere in the execution of the routine. The tests reside in a class that inherits from ES_TEST, and the tests are added to the database (using the Eiffel agent mechanism) either with an *add_boolean_case* or *add_violation_case* declaration. If a specific contract must be checked, the associated violation tag can be declared using an *add_violation_case_with_tag* declaration.

```
class CUSTOMER_AND_RULE_TESTS inherit ES_TEST create
    make
feature {NONE}
  make is
    do
      add_boolean_case (agent test_a_bad_rule1)
      add_violation_case_with_tag ("r1", agent test_a_bad_rule2)
    end
    -- More agents here

 test_a_bad_rule2  is
    local
      bob: CUSTOMER
      a_rule: RULE
    do
      create bob.make ("Bob", 5999, true, 12)
      create a_rule.make (bob, -1, 6000, 1000)   -- contract violation
    end
end
```

Figure 20. Violation case for incorrect rule

6. Flexible Fixture redefinition

ES-Fit supports the three standard Fit Fixtures: ES_COLUMN_FIXTURE, ES_ROW_FIXTURE and ES_ACTION_FIXTURE. In addition to standard Fixtures, ES-Fit provides a flexible mechanism to develop new Fixture types. We first discuss the standard Fixtures and then describe how to create new Fixture types. Column Fixtures were already explained and illustrated earlier (see Figure 1 and Figure 4).

Calculate Credit Action 1		
start	Credit Database	
enter	name	Bob
enter	balance owing	5000.00
enter	reliable	true
enter	months trading	14
press	add to database	
check	count	1
enter	name	Anne
enter	balance owing	6000.00
enter	reliable	true
enter	months trading	18
press	add to database	
check	count	2

Figure 21. Action Fixture to add customers to the database

6.1. Action Fixture

An Action Fixture, associated with a table, tests that a series of actions carried out on an application works as expected [12]. An Action Fixture starts a named class by creating an actor which is an instance of that class. Subsequent actions are made through feature calls on that separate actor object. The allowed actions (in addition to *start*) are *enter* (setting an attribute), *press* (calls a command) and *check* (can be used to check the value of a query). The Fit developer can create an Action Fixture by inheriting from ES_ACTION_FIXTURE.

We can use our credit example to illustrate an Action Fixture. We may wish to add some new customers (Bob and Anne) to a Credit Database. As shown in Figure 21, our user may specify these actions as follows:

- The *start* action in the first row creates an object associated with the Credit Database.
- The *enter* (name) action in the second row calls a routine residing in CREDIT_ACTION_FIXTURE that is bound to the item in the second column ("name"). The developer may choose any routine to be bound to name (e.g. *set_name*) passing to it the string "Bob" in the third column.
- After entering some more information about Bob, such as his balance owing, months trading, and other information, our user may specify *press* ("add to database") in the sixth row of Figure 21, to indicate that Bob's data is complete and he can be added to the database.

- The *check* (*count*) in the seventh row can be used to check that the expected value of *count* (i.e. one item in the database) matches with the actual database. If there is a match, then the check field will be marked with green and this Fit test passes.

The (partial) Fixture code associated with the Action Fixture table is shown in Figure 22. The developer must implement the deferred routine *start* when inheriting from ES_ACTION_FIXTURE.

```
class CREDIT_ACTION_FIXTURE inherit ES_ACTION_FIXTURE  create
  make
feature {NONE}
  make is
    do
      bind ("name", agent set_name)
      bind ("balance owing", agent set_balance)
      bind ("add to database", agent add_to_database)
      -- More bindings in here ...
    end

  a_customer: CUSTOMER
  name: STRING
  balance: REAL
  reliable: BOOLEAN
  months_trading: INTEGER
  customer_database: CREDIT_DATABASE

  set_name (a_name: STRING): TUPLE is
    do
      name := a_name
    end

  set_balance (b: REAL): TUPLE is
    do
      balance := b
    end

  start is
      -- starts the application
    do
      create customer_database.make
    end

  add_to_database: TUPLE is
      -- set the telephone number of the object
    do
      create a_customer.make (name, balance, reliable,
                                                  months_trading)
      customer_database.add (a_customer)
    end

  -- More methods in here ...
end
```

Figure 22. Code for Credit Action Fixture

6.2. 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) [12]. We can use a Row Fixture table (Figure 23) to check that Bob and Anne are in the database. The associated Row Fixture code is shown in Figure 24. The developer creates a Row Fixture (say, CREDIT_ROW_FIXTURE) by inheriting from ES_ROW_FIXTURE[G], where G is a generic parameter which must be instantiated to the type of the object in the database (in this case CUSTOMER).

Calculate Credit Row 1					
name	balance owing	reliable	months trading	Allow Credit*	Credit Limit*
Bob	5000.00	true	14	true	1000
Anne	6000.00	true	18	false	0.00

Figure 23. Table associated with the Row Fixture

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[G] representing the items in the database of the business logic. In our case, we need to connect the Row Fixture with the Action Fixture (Figure 22), i.e. the *query* routine should return the database listing from the Action Fixture object.

6.3. Defining new Fixture types

ES-Fit provides an easy mechanism for the developers to redefine the behaviour of the standard Fixtures (i.e., Column, Row and Action) in order to produce new types of Fixtures. The ES-Fit tool abstracts away unnecessary loop structures from the Fixture code. This is due to the fact that the underlying ES-Fit engine reads and processes one row at a time. This abstraction is supported by class ES_FIXTURE_UNIT (the ancestor of all Fixtures) which treats all tables row by row, hence enabling easy redefinition. Therefore, Fit developers only need to define how ES-Fit should process one row at a time, instead of defining how to process the complete table. The following routines in ES_FIXTURE_UNIT provide this flexible redefinition capability:

- *pre-process-table* (actions before reading a table)
- *pre-process-row* (actions before reading a row)
- *process-row* (how to process a row)
- *post-process-row* (actions after reading a row)
- *post-process-table* (actions after reading a table)

Consider, for example, the table in Figure 25. The table looks like a Row Fixture, but it also has a *Total Credit* in the last row (as in a spreadsheet). Since the table is very similar to a Row Fixture, we may inherit from ES_ROW_FIXTURE and redefine the standard behaviour (Figure 26). Routines *process_row* and *post_process_table* are redefined.

Routine *process_row* is instructed to ignore the last row, as this row will be treated in *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 checked.

```
class CREDIT_ROW_FIXTURE inherit ES_ROW_FIXTURE [CUSTOMER]
create
  make
feature {NONE}
  make is
    do
      bind("name", agent get_name)
      bind("balance owing", agent get_balance)
      -- More bindings here ...
    end

  get_name(a_customer: CUSTOMER): TUPLE is
      -- returns the name of the customer
  do
    Result := [a_customer.customer_name]
  end

  get_balance(a_customer: CUSTOMER): TUPLE is
      -- returns the balance of the customer
  do
    Result := [a_customer.balance]
  end

  -- Other methods in here ...

  query: LINKED_LIST[CUSTOMER] is
    local
      action_fixture: CREDIT_ACTION_FIXTURE
      database: CREDIT_DATABASE
    do
      action_fixture ?= connected_to
      database := action_fixture.customer_database
      Result := database.customers
    end
end
```

Figure 24. Partial code for Credit Row Fixture

In *post_process_table*, we must (a) specify the cell where the total credit will be printed, and (b) bind a routine to the cell (in this case routine *add_credit*). We may think of the table as a spreadsheet with the appropriate cell at the intersection of row *Total Credit** and column *Credit Limit** (Figure 25). The specification of this cell is done by a call to *connect_to_target* (inherited from ES_FIXTURE_UNIT). As before, the binding of the cell to routine *add_credit* is done in the constructor *make*. The *add_credit* routine calculates the required totals with a suitable call to the business logic.

7. Unification of Unit Tests and Fixtures

In Java, a developer may use JUnit for Unit Tests and the Fit command line application for the table acceptance tests. 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 specification will be reported in the Unit Tests and the Fit Tables. The second addition is that we unify the Fit Fixtures and Unit Tests in the same class, so that validation and verification can be performed simultaneously in order to certify the quality of the product.

Calculate Credit Row 2					
name	balance owing	reliable	months trading	Allow Credit*	Credit Limit*
Bob	5000.00	true	14	true	1000
Anne	6000.00	true	18	false	0.00
				Total Credit*	2000.00

Figure 25. A new Fixture Type

```
class CREDIT_ROW_FIXTURE2 inherit ES_ROW_FIXTURE [CUSTOMER]
    redefine process_row, post_process_table end
create
  make
feature {NONE}
  make is
    do
      bind("name", agent get_name)
      -- More bindings in here ...
      bind("Total Credit*", agent add_credit)
    end

  get_name(a_customer: CUSTOMER): TUPLE is
    do
      Result := [a_customer.customer_name]
    end

  add_credit: TUPLE is
    do
      Result := [database.total_credit]
    end

  -- additional methods such as query as in Figure 24

  process_row is
      -- redefined: ignores the last row
    do
      if not (content_under_heading ("Allow 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*", "Credit Limit*")
      execute_cell ("Total Credit*")
    end
end
```

Figure 26. Code for a redefined Row Fixture

In order to run the Fit tests, the software developer places the test Fixtures in a class that inherits from class ES_FIT. Unit Tests are placed in a class that inherits from ES_TEST. We may combine Validation (Fit Fixture Tests) and Verification (Unit Tests and Test Specifications) under the report of a single green bar by declaring both types of tests in a class that is a descendant of ES_SUITE. Eiffel supports multiple inheritance, and thus ES_SUITE unifies the two types of tests in a single class by inheriting from both ES_FIT and ES_TEST (Figure 27 and Figure 28).

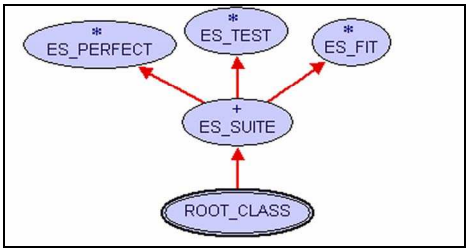


Figure 27. Inheritance hierarchy of ES_SUITE

```
class ROOT_CLASS inherit
  ES_SUITE
create
  make
feature -- Initialization

make is
    -- Creation procedure
  do
    add_fixture ("Calculate Credit 1", create{CREDIT_FIXTURE}.make)
    add_fixture ("Calculate Credit 2", create{CREDIT_FIXTURE2}.make)
    add_fixture ("Calculate Credit 3", create{CREDIT_FIXTURE3}.make)
    add_test (create{CUSTOMER_AND_RULE_TESTS}.make)
    run_espec -- runs all specifications
  end

end -- class ROOT_CLASS
```

Figure 28. Root class of the combined system

The software developer may run both types of tests simultaneously under a single green/red bar via the *Run all Specs* button in ESpec tool (Figure 3). Alternatively, the developer can run a specific type of tests by invoking the associated button (*Run ES-Test* and *Run ES-Fit*). Test results for both types of tests are reported in the tool results window.

```
class RULE create
  make
feature
  customer: CUSTOMER
  max_credit_allowed, max_balance_owing: REAL
  min_months_trading: INTEGER

  make (a_customer: CUSTOMER; min_months: INTEGER;
        max_balance, max_credit: REAL) is
    require
      r1: min_months >= 0 and max_credit >= 0.0 and
          max_balance >= 0.0 and a_customer /= Void
    do
      customer := a_customer; min_months_trading := min_months
      max_credit_allowed := max_credit;
      max_balance_owing := max_balance
    ensure
      e1: customer = a_customer and
          min_months_trading = min_months
      e2: max_credit_allowed = max_credit and
          max_balance_owing = max_balance
    end

  allow_credit: BOOLEAN is
    do  -- implementation bug: '<=' should be '<' at line 1
      Result := customer.balance <= max_balance_owing
      Result := Result and customer.is_reliable and
                customer.months_trading > min_months_trading
    ensure
      e1:  Result = (customer.balance < max_balance_owing and
           customer.is_reliable and
           customer.months_trading > min_months_trading)
    end

  credit_limit: REAL is
    do
      if customer.balance < max_balance_owing  and
        customer.is_reliable  and
        customer.months_trading > min_months_trading
      then
        Result := max_credit_allowed
      else
        Result := 0.00
      end
    ensure
      e1: allow_credit  implies Result = max_credit_allowed
      e2: not allow_credit implies Result = 0.0
    end
invariant
  customer_not_void: customer /= Void
end
```

Figure 29. Contracts for class RULE

8. Formal verification [ES-Perfect]

Starting with the informal requirement [R1], we developed Fit acceptance tests from the point of view of our client. We developed the code starting with Early Testable Specifications which resulted in the business logic package involving classes CUSTOMER, RULE, and CREDIT_DATABASE. In each case, contract violations were reported, either in the Fit Tables or in the Unit Tests. These Validation and Verification tests are relentlessly concrete and provide a certain amount of assurance that the code is correct. These checks are all performed dynamically at run time.

In order to provide a higher level of certification, the ESpec tool is equipped with an experimental formal verification module called ES-Perfect. The tool (invoked from ESpec, see Figure 3) takes as input a package (e.g. the set of classes in the business logic) and transforms the classes into predicate logic assertions in the *Perfect Language* [5]. The predicates in this language can be verified by applying the Perfect Developer theorem prover[2]. Thus, each class is checked for correctness. This means, that for each routine in the class, the implementation must be shown to satisfy the specification (written via the pre-conditions, postconditions and invariants).

As an example, consider class RULE (Figure 29) which is equipped with complete contracts. ES-Perfect provides feedback that the postcondition of RULE.*allow_credit* fails. An examination of the implementation of this routine indicates that there is an error in the first line, which should be changed to:

```
Result := customer.balance < max_balance_owing
```

With this code fix, class RULE is certified correct with 6 verification conditions. Class RULE is simple enough that it verifies almost instantaneously. However, The ESpec translator can currently deal with more complicated constructs including generic types, loops, quantifiers, and all contracts including preconditions, postconditions, class invariants and loop variants and invariants. We are currently extending the tool to deal with inheritance and complete reference semantics.

The Perfect language and theorem prover is a formal tool aimed at software development and verification. The theorem prover is completely automatic. This is important because it means that the software developer can use it without any knowledge of the formal underpinnings. If a verification cannot be proved, the theorem prover will report this back to the user who can then look into the reason for the failure

[2] A version of the Perfect Developer theorem prover is incorporated in ESpec with the permission of Escher Technologies.

(a failure does not necessarily mean a bug as it may just be that the prover was not sufficiently powerful).

Perfect Developer is built around a predicate logic based notation for expressing state-based specifications and optionally refining them to a form resembling a program (as in the Z or B method [2, 15]). The Perfect language has a library of useful collections and structure types such as sets, bags, sequences, and maps [4]. In order to translate from the Eiffel contract language to Perfect, we therefore developed an equivalent mathematical library in Eiffel (to be reported on in a future paper).

9. Conclusions

In this paper we provided a lightweight formalization of Validation and Verification with the addition of Design by Contract. We used Fit Tables for answering the Validation question and Early Testable Specifications for answering the Verification question. Early Testable Specifications involved the use of Test Driven Development (Unit Tests) as well as Design by Contract. Contract violations were signaled in Unit Tests as well as Fit Tables. We argued that V&V can be specified in a testable format very early in the software life cycle, and the working product is subjected to both Validation and Verification tests right from the very beginning. Any change to the software product must be regression tested against both the Validation and Verification tests, thus providing some guarantee for a quality product. We provided the ESpec tool, the first implementation of the Fit framework in the Eiffel programming language that embodies some of these fundamental ideas for formal V&V throughout the lifecycle, and aids in the achievement of product quality certification written in Eiffel. The tool extends the Fit framework with some additional features that aid in formalizing customer requirements, including the ability of a table to reference data in other tables, flexible naming conventions, and a flexible method for constructing new Fixture types. ES-Perfect combines classical ideas of formal specification and verification with these new automatic checking technologies. As shown in Figure 27, ESpec allows a developer to mix all three types of tests, i.e. Fit Tests, Unit Tests and formal verification in a single class that inherits from ES_SUITE. The tests can be run individually or in unison with feedback to a single green/red bar. The combination of all these checks thus results in a higher quality of certification.

10. Acknowledgments

ESpec project made possible by the collaborative inputs and contributions of all members at the Software Engineering Lab, York University. Our special thanks to Eric Kerfoot for developing ESpec's Eiffel parser (ES-Parser) and Jackie Wang for developing the Eiffel to Perfect translator.

11. References

[1] "IEEE Standard for Software Verification and Validation Plans," IEEE Std 1012-1986, 1986.

[2] J.-R. Abrial, *The B-Book: Assigning programs to meanings*: Cambridge University Press, 1996.

[3] K. Beck, *Test-driven development: by example*. Boston: Addison-Wesley, 2003.

[4] G. Carter, R. Monahan, and J. Morris, "Software Refinement with Perfect Developer," presented at Software Engineering and Formal Methods Conference (SEFM2005), 2005.

[5] D. Crocker, "The Perfect Developer Language Reference Manual: Version 3.0," Escher Technologies 2004.

[6] W. Cunningham, "Framework for Integrated Test, Java Platform, http://fit.c2.com/wiki.cgi?JavaPlatform," 2005.

[7] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A Reference Model for Requirements and Specifications," *IEEE Software*, vol. 17, pp. 37-43, 2000.

[8] M. Jackson, *Software Requirements and Specifications*: Addison-Wesley, 1995.

[9] A. Leitner, "Erl-G: Eiffel Reflection Library Generator," 2005.

[10] B. Meyer, *Object-Oriented Software Construction*: Prentice Hall, 1997.

[11] Mike Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview," in *CASSIS 2004*, vol. LNCS 3362: Springer Verlag, 2004.

[12] R. Mugridge and W. Cunningham, *Fit for Developing Software: Framework for Integrated Tests*: Prentice-Hall, 2005.

[13] J. S. Ostroff and R. F. Paige, "The Logic of Software Design," *Proc. IEE - Software*, vol. 147, pp. 72-80, 2000.

[14] K. Rustan, M. Leino, G. Nelson, and J. B. Saxe, "ESC/Java User's Manual," Compaq Systems Research Center Technical Note 2000-002, October 2000.

[15] J. M. Spivey, *The Z Notation: A Reference Manual (2nd edition)*. Englewood Cliffs, N.J.: Prentice-Hall, 1992.