Invited paper, Tests and Proofs - TAP2007, ETH Zurich, February 2007, LNCS 4454 ("Tests and Proofs").

Testable Requirements and Specifications

Jonathan S. Ostroff and Faraz Ahmadi Torshizi

Department of Computer Science and Engineering, York University,
4700 Keele St.,Toronto, ON M3J 1P3, Canada
{jonathan, faraz}@cse.yorku.ca

Abstract. A design *specification* is the artifact intermediate between implemented code and the customer *requirements*. In this paper 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. The core idea behind early testable requirements is that the problem is described before we search for a solution that can be tested against the problem description. We also want the problem description to drive the design. We provide a method for describing early testable requirements and specifications and a support tool called ESpec. ESpec allows for the description of testable requirements via Fit tables as well as testable design specifications via contracts written in Eiffel using mathematical models following the single model principle. The tool can mechanically check the requirements and specifications.

1 Introduction

Informal surveys such as those done by the Standish Group [5] show that a minority of software development finish on time and within budget. Many projects fail entirely and have to be abandoned. In their recipe for success the Standish group recommends that shareholders develop the ability to clearly articulate requirements and translate these requirements between the business people (the customers) and the technical people (software developers).

The software developer faces many difficulties in writing and communicating requirements. As one IT specialist wrote [2]:

> 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 communication 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.

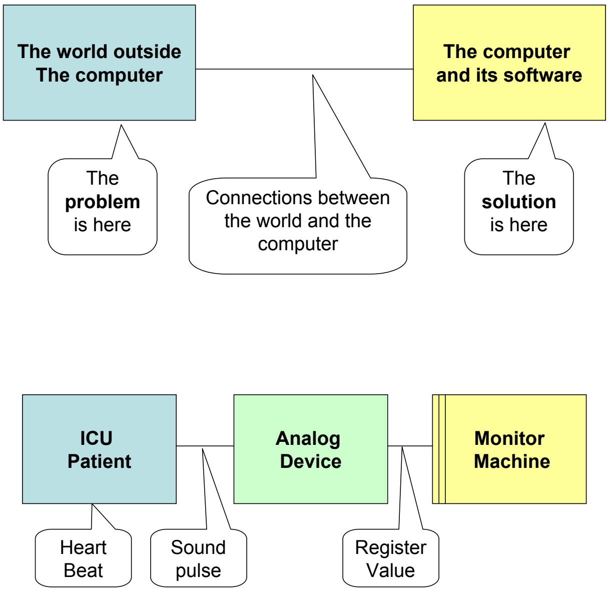


Fig. 1. The Computer (the Machine) and the World Outside the Computer (Problem Domain)

In this paper we use the words "requirements" and "specifications" in the sense of Jackson [7]. 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 provide a method for describing early testable requirements and specifications and a support tool called ESpec. ESpec allows for the description of testable requirements via Fit tables adapted from [8] for Eiffel. Testable design specifications are described via contracts written in Eiffel using mathematical models following the single model principle. The tool can mechanically check the requirements and specifications.

2 Requirements and Specifications

Consider the diagram in Fig. 1 illustrating the problem of measuring vital signs such as the heartbeat of a patient in an ICU taken from [7]. There are four different descriptions of the patient monitoring system.

P – Problem Domain: A patient's heart can beat from 0 to 170 beats per second (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**.

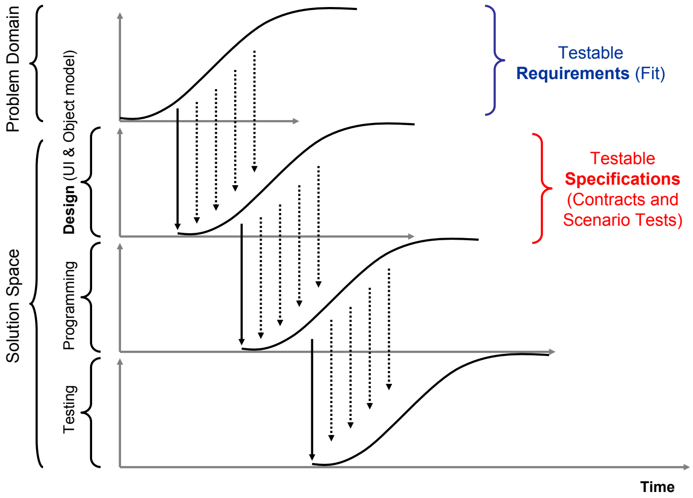


Fig. 2. The role of Early Testable Requirements and Specifications in the design cycle

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 satisfies requirements **R**. Finally, **C** is a description of the computer code needed to implement the design specification **S**. As mentioned earlier, the design specification is the artifact intermediate between implemented code and the customer requirements.

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. We provide a method for describing early testable requirements and specifications and a support tool called ESpec.

ESpec allows for the description of testable requirements via Fit tables adapted from [8] for Eiffel. Testable design specifications are described via contracts written in Eiffel using mathematical models following the single model principle. The tool can mechanically check the requirements and specifications.

Our method and tool does not require a rational software development process as described earlier. The developer may follow agile [1] or big design up front methodologies. Our method does provide a framework for describing testable requirements that can be written as early as possible in the design cycle as shown in Fig. 2.

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 [10]:

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 (specification correctness) asserts that we are developing the right product, i.e. the one desired by the customer as described by R. The second equation (implementation correctness) asserts that the product is being developed correctly, i.e. the implemented code satisfies the design specification.

The third formula (system correctness) is a consequence of formulas (1) and (2). It asserts that the code working in a problem domain P satisfies the customer requirements.

3 Fit Tables as Testable Requirements

How do we make requirements testable? In this section we show how Fit tables may be used to make requirements testable early in the design cycle. We use a small application as a running example in this section and the sequel.

3.1 Informal requirements for a chat application

Suppose our customer is a company that needs a specialized chat application allowing employees to communicate with each other. Chat rooms can be developed for technical support or discussions on various administrative issues. Some of the informal requirements include:

[R1] A chat server has an Administrator and a public room called the Lobby.
[R2] A user may connect to the chat server initially landing in the Lobby.
[R3] A user may add or remove public or private rooms thus becoming the owner of that 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.

The requirements are expressed in terms of the phenomena of the problem domain such as chat rooms like the Lobby, users such as the Administrator and owner relationships between users and rooms. Phenomena of the solution domain such as linked lists of users or binary search routines for finding users in the lists should not be part of the requirements.

3.2 A Fit table to test the first requirement

How do we convert the informal requirements into testable requirements? We can make the first requirement testable with the simple Fit table shown in Table 1.

As described in [8], there are three basic tables types: Column, Action and Row. A testable requirements document may contain informal text interleaved with an arbitrary number of Row, Column and Action tables.

For requirement R1 we use an Action table. An Action table checks that a sequence of actions performed on an application works as expected. In the sequel we will also see examples of Row tables. Software developers may also use the Fit framework to specify their own table types.

In the first row of Action table 1 the Customer provides an arbitrary title such as: "R1: Chat Server Setup". In the first column of the table we can see keywords (**start**, **check**, **enter** and **press**) which denote the type of action performed by each row.

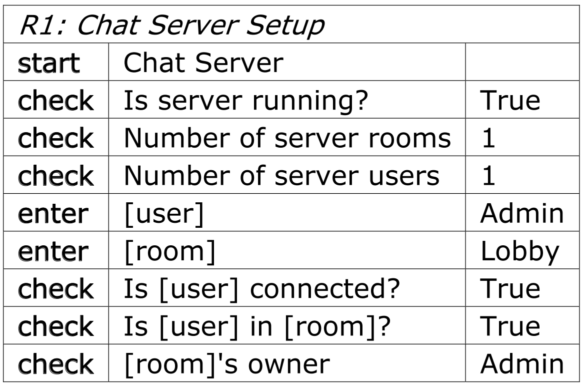
The keyword **start** is used to initiate the chat server. Usually there is only one **start** per Action table. Thus the second row of the table starts the business logic for the chat server. The next Action table in the same document will use the current chat server unless there is another **start**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

The keyword used in the third row is **check**. It checks 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 "Is server running?" must have the value "True".

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. In Action table 1, "True" is a boolean, "1" is an integer, and "Admin" is a string. As far as the customer is concerned, a value is just a descriptive string.

Consider the check for the property "Is [user] in [room]?" in row 9 of Action table 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

- 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”.



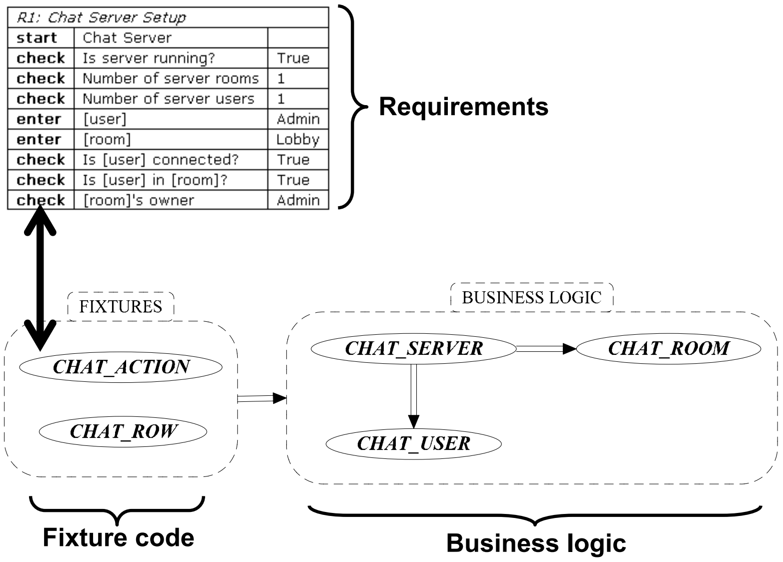
R1: Chat Server Setup		
start	Chat Server	
check	Is server running?	True
check	Number of server rooms	1
check	Number of server users	1
enter	[user]	Admin
enter	[room]	Lobby
check	Is [user] connected?	True
check	Is [user] in [room]?	True
check	[room]'s owner	Admin

Table 1. Chat Action table for requirement R1

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 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. 13. This means that user “Bob” adds the room “Technical Support” to the chat application, and “Bob” is now the owner of the room.

How does the developer satisfy the requirements specified in the Action table? The developer will need to write two kinds of classes: *Fixture* classes and classes of the business logic (see Fig. 3). Fixture code acts as a glue code or bridge between the customer-provided requirements and the business logic. The ESpec tool provides fixture libraries that allow the developer to easily develop such fixture classes. Fit framework can then use the developer written fixture classes to parse the requirements document, extract the Row Column and Action 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 that 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

Fig. 3. Relationship between Fit tables, Fixture classes and Business logic

For example, to run Table 1, the developer writes an Action Fixture class `CHAT_ACTION` that binds developer defined routines (in the business logic) to the properties in the table. These routines call the appropriate features of the business logic. ESpec takes care of the rest of the processing as explained in more detail in [14].

3.3 Implementation Correctness

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 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).

```
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
```

Fig. 4. Scenario Test to start a server, connect users, create a room and set the room permissions

How does the developer specify implementation code that will satisfy the requirements? It is unlikely that the code can be developed all at once. The requirements are described at a relatively high level in terms of the phenomena of the problem domain. Code will have to be developed in small chunks to build up the functionality needed to provide a solution. This functionality is the *design* which is intermediate between the code and the requirements. The Fit table requirements were expressed in terms of the phenomena of the problem domain. The design will be specified in terms of the phenomena of the machine (solution space), i.e. we must specify the relevant classes and features that will solve the problem posed by the requirements.

How do we specify testable designs? We will use a combination of Contracts and Scenario Tests to specify the design. In this section we illustrate Scenario

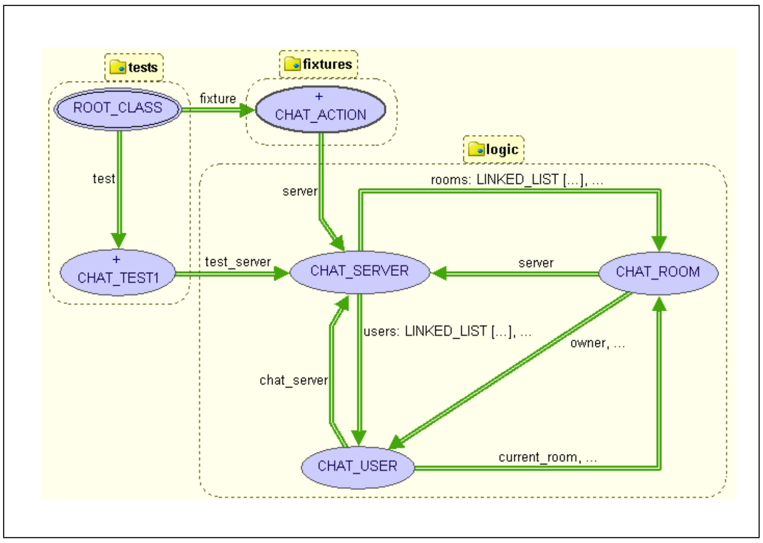


Fig. 5. Design of the chat application as a BON class diagram

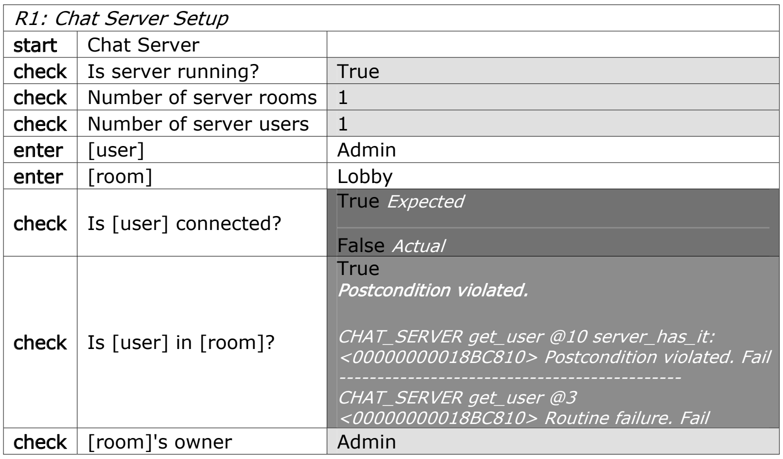
Tests and in the next section we will use Contracts. This should not be taken as a description of a step-by-step software development methodology. In the actual development, developers may use any combination of coding, Scenario Tests, contracts and other development techniques in whatever order they choose. Our contribution is to provide a method and tool for specifying designs as early as possible in the design cycle and mechanically testing implementations against the design specification.

Consider the Scenario Test in Fig. 4 expressed in the unit testing framework developed for Eiffel [11]. A Scenario Test is written the same way as a unit test but instead of testing only one unit of functionality, it tests the collaboration between various elements in the business logic. The Scenario Test in Fig. 4 specifies a collaboration between classes `CHAT_SERVER`, `CHAT_ROOM`, and `CHAT_USER`. The test specifies specific features in `CHAT_SERVER` such as:

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

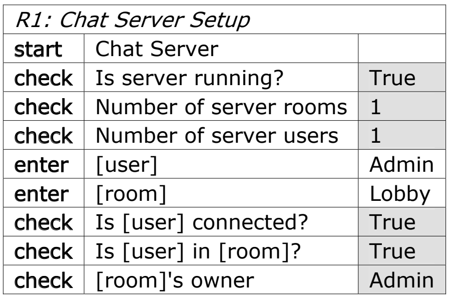
If all the classes and features in the Scenario Test are added, the project will compile and the design illustrated in the BON class diagram in Fig. 5 is generated automatically. The class diagram presents the design so far (classes and feature signatures, but not yet code in the bodies of the features).

The ESpec quality workbench will run all the Scenario Tests and show which ones fail with a red bar. The Scenario Test will fail if (a) the collaboration between the various elements fails to satisfy the specified checks or to produce



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 2. Result of running Table 1 with implementation or contract errors in the business logic. Light grey indicates tests that succeed (green) and dark grey tests that fail (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

Table 3. Result of running Table 1 after fixing the business logic. All tests succeed (light grey = green)

the anticipated results, or (b) the contracts fail while executing the tests. At this point we have not specified any contracts so failures will be of type (a).

Scenario Tests (as in Fig. 4) thus do two things for us: (1) They specify the design, in a (2) mechanically testable manner. Contracts will likewise specify aspects of the design. With runtime assertion checking turned on the implementations are also checked against the contracts.

There is thus a synergy between the contracts and Scenario Tests. They both specify aspects of the design and both are mechanically checkable. Contracts act as test amplifiers, i.e. when we execute the tests, all contracts will also be executed and tested.

3.4 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!

How do we test that our specified design satisfies the requirements? We do this by hooking up the Fit tables to our business logic for the chat application and running the Fit table tests. If we run Action table 1 with an incorrect implementation or design, we get error results of the kind shown in Table 2.

Two types of errors are shown in Table 2. The first error (row 8 shown in dark gray) 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"). We refer to these as *category 1 errors.* 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 gray) indicates a postcondition failure in the business logic (in `CHAT_SERVER.get_user`). We refer to these type of errors as *category 2 errors.* Category 2 errors usually indicate an implementation problem in the business logic (i.e., the implementation failed to satisfy its contracts).

We will discuss the differences between these types of errors in the sequel. The result of running Action table 1 (after all fixes) is shown in Table 3. All the cells in the table are shown in light gray (green) indicating that all the Fit tests succeed.

4 Writing Complete Modular Contracts

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

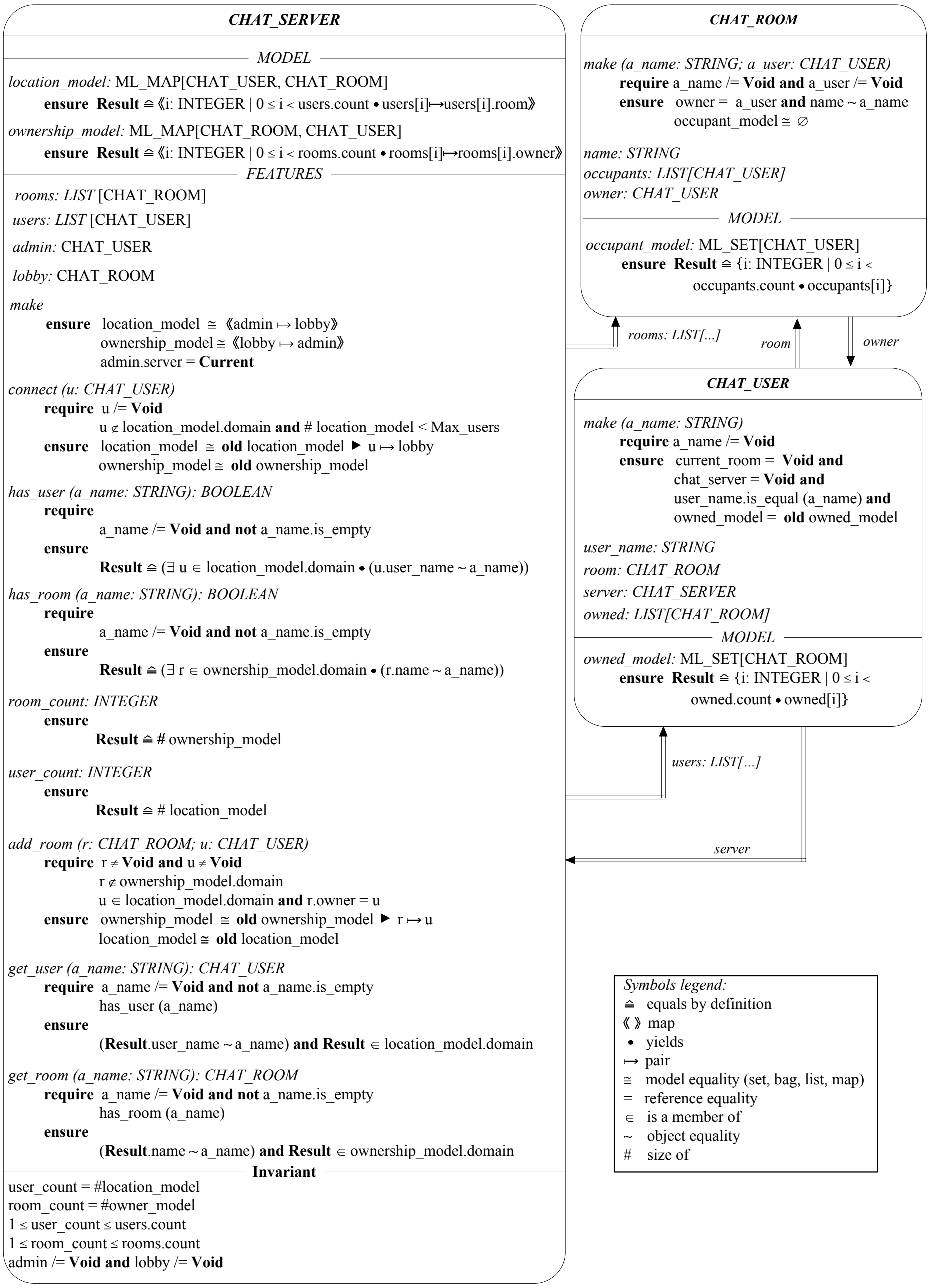
```
class CHAT_SERVER ...

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
```

Fig. 6. Incomplete contract for routine `connect`

Design by Contract (DbC) is a well-know method for specifying the obligations and benefits of the client of a module (class) and its supplier. Languages such as Eiffel, ESC/Java [4] and Spec# [3] follow the single model principle [12], i.e. specifications (contracts) and implementation details are both an integral part of the program text itself thus also allowing the implementation to be mechanically checked against the specification. The features of a class are described by expressive preconditions, postconditions and class invariants and these contracts can be tested at runtime by checking that the feature implementations satisfy the contracts.

However, the contracting facilities of these languages do not yet allow for complete contracts. We illustrate this lack and describe the use of an implemented mathematical modelling library (ML) for Eiffel that facilitates fully descriptive contracts following the single model principle so that the full contracts are part of the program text.

Consider the contract for the routine `connect` in class `CHAT_SERVER` in Fig. 6. 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



```
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

Fig. 7. System Specifications in BON notation

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.

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?

4.1 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 describe the above state of affairs. The *location model* is a function from `CHAT_USER` to `CHAT_ROOM` as shown in Fig. 8.

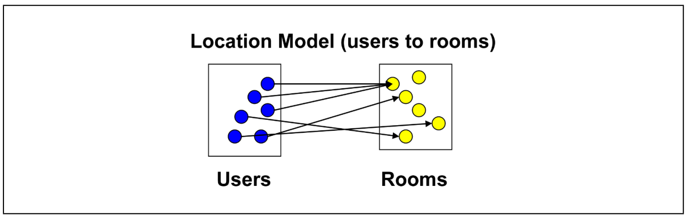


Fig. 8. Location model – mapping from users to rooms

In the location model each user is associated with a room. The location model may be described using the mathematical map class `ML_MAP` in the ESpec model library (these classes all have the prefix `ML`) as shown in Fig. 9. The ML classes are immutable. Thus they have no commands that can change their state, only queries that may return new maps constructed from the old maps as in mathematics. These classes are thus mathematically expressive but not efficient. This is not a problem as models are used solely in contracts and contract checking can be turned off in final deliveries.

In Fig. 9, the location model is specified as

(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
            $location\_model \cong \textbf{old}\ location\_model \blacktriangleright u \mapsto lobby$
            $ownership\_model \cong \textbf{old}\ ownership\_model$
        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 ˆ [u, lobby]
            ownership_model |=| old ownership_model
        end
end
```

Fig. 9. Complete contracts for routine `connect` using a mathematical model

location_model: ML_MAP[CHAT_USER,CHAT_ROOM]

The 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 postcondition is

$$location\_model \quad \cong \quad (\textbf{old}\ location\_model) \blacktriangleright (u \mapsto lobby) \tag{1}$$

which asserts that after execution of `connect`, the *location_model* is extended by (symbol $\blacktriangleright$) 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 $\cong$ is the *model equality* symbol. Two maps are 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 [13] mathematical notation as in (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. 9(b) the Eiffel equivalent of (1) is

location_model |=| **old** location_model ˆ [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 [9]. For example, the postcondition of query `has_user` in Fig. 12 is specified as

$$Result \quad \widehat{=} \quad \exists\, u \in location\_model.domain \bullet (u.user\_name \sim a\_name) \tag{2}$$

The symbol $\sim$ denotes object equality. The 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_user`.

An implementor of class `CHAT_SERVER` may provide any private implementation code that satisfies the 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 [6] that maps (or "lifts") the implementation detail to the model as shown in Fig. 12. For example for the location model, the abstraction function is

$$Result \mathrel{\widehat{=}} \langle\!\langle i : INT \mid 0 \leq i < users.count \bullet users[i] \mapsto users[i].room \rangle\!\rangle \tag{3}$$

The angle brackets $\langle\!\langle \cdots \rangle\!\rangle$ is used for map comprehension (similar to set comprehension). Thus (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. 11. 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. 7). 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. 7.

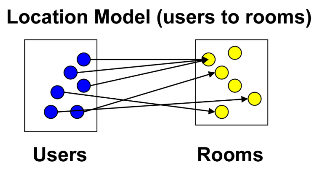


Fig. 10. Location model (mapping from users to rooms)

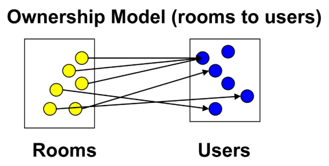
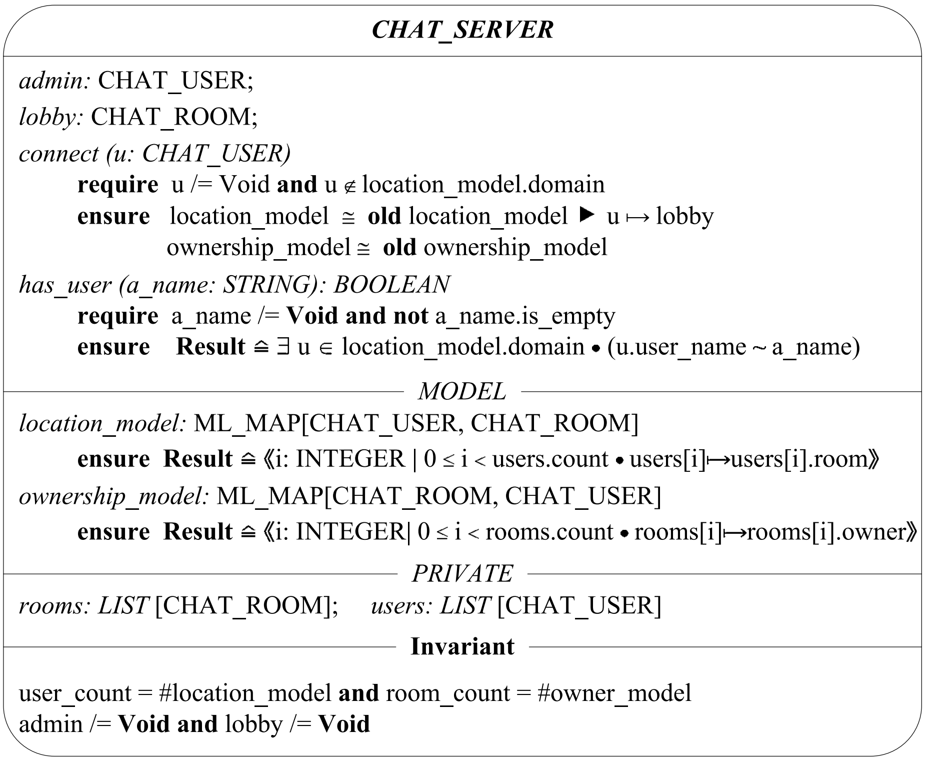


Fig. 11. Ownership model (mapping from rooms to users)



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**

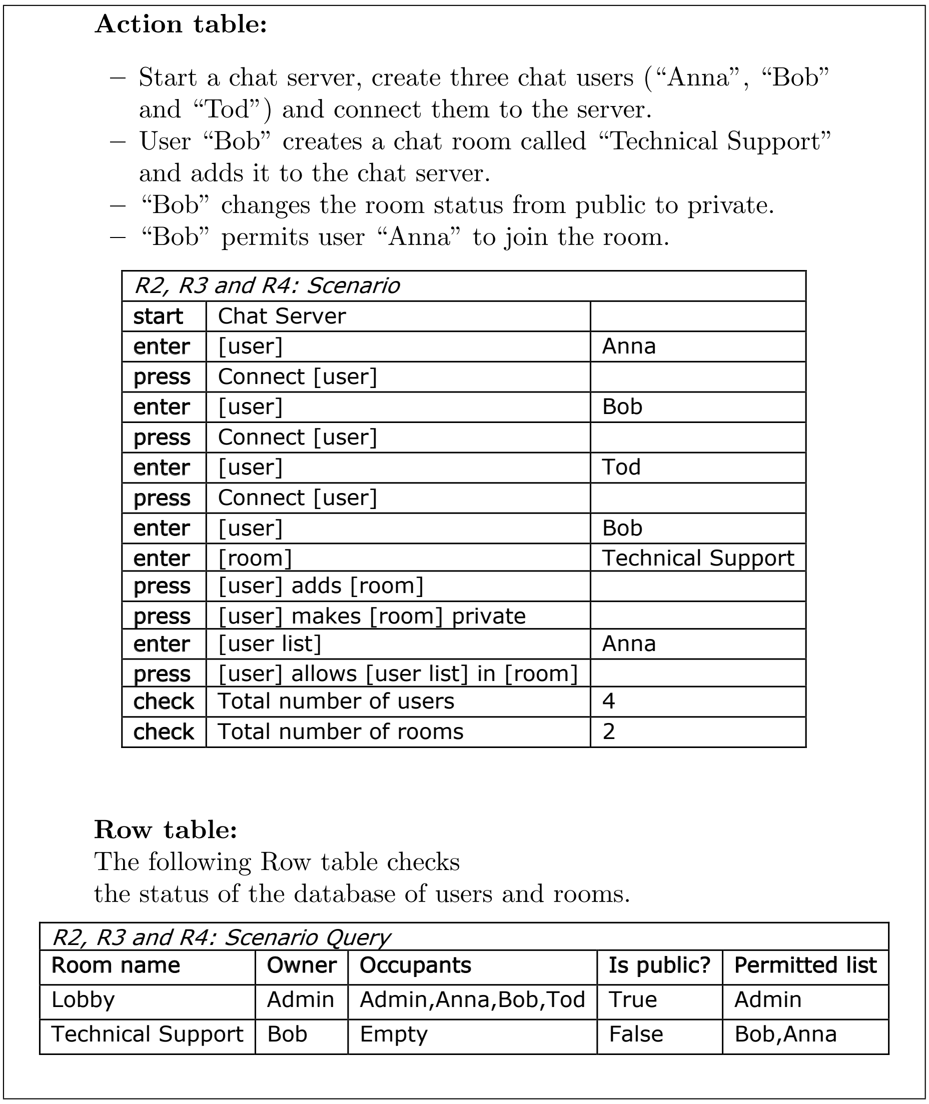
Fig. 12. BON specification of the chat server

5 Contract violations in Fit tables

Section 3.3 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 used contracts to specify the design. The advantage of contracts (as opposed to Scenario Tests for a particular execution) is that contracts specify the complete behaviour of modules (classes). In ESpec, 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. 13 and Fig. 14. These tables convert requirements R2, R3, and R4 into a mechanically testable format.



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

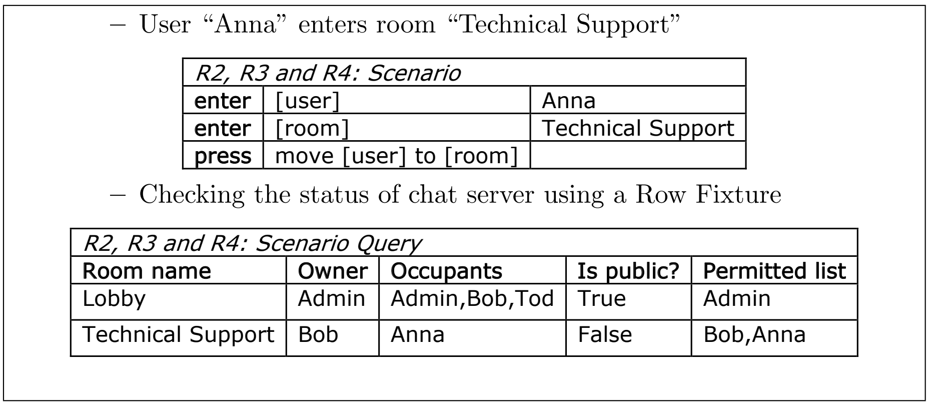
Row table:
The following Row table checks
the status of the database of users and rooms.

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

Fig. 13. Testing requirements R2 – R4

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. 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.



- 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

Fig. 14. Moving a user from one room to another

Fig. 13 has an Action table and a Row table. The Action table starts a chat server, and adds users, rooms and permissions as shown. Consider the Row table in Fig. 13. 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.

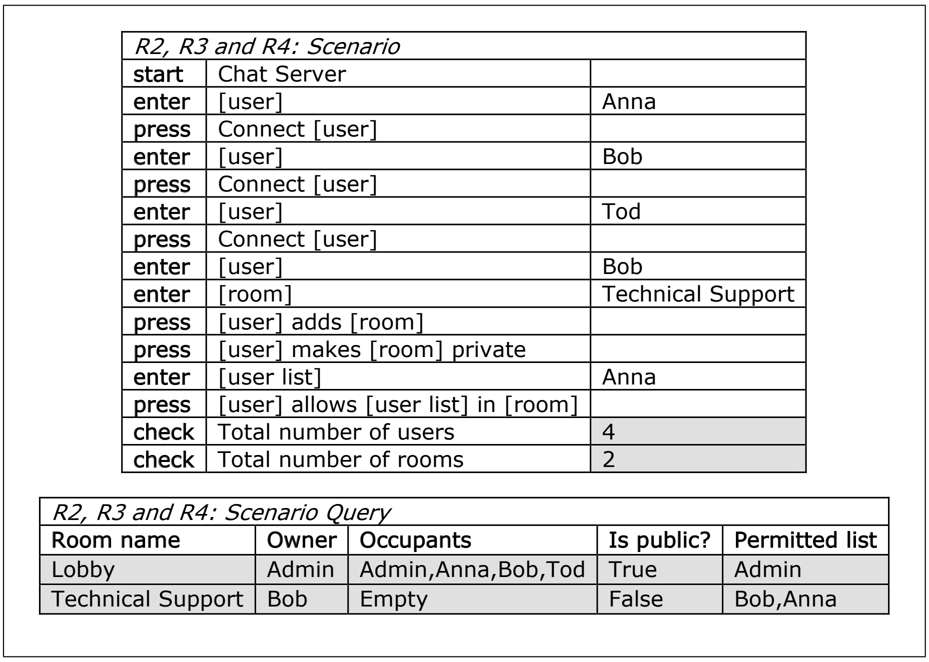
It is of course up to the developer to connect the Row table to the business logic via a Row fixture. In our case, the developer uses the fixture to link the Row table to `rooms` in the business logic which is a linked list of `CHAT_ROOM`. The linked list and class `CHAT_ROOM` are of course phenomena in the machine.

Each row in the Row table describes the properties of a room in the collection. In the Row table of Fig. 13 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 represent *exhaustive* descriptions of the collection.

The other column headings of the Row table in Fig. 13 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.

Fig. 14 is a continuation of the requirement document described in Fig. 13, i.e. it refers to the same chat server initialized and acted upon in Fig. 13. Subsequent to the actions of Fig. 13, our customer uses the Action table in Fig. 14 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 Support.

Do the Fit tests pass given the design developed in previous sections of this paper? If we execute the Fit requirement tests described in Fig. 13 and Fig. 14 we obtain the results shown in Fig. 15 and Fig. 16.



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

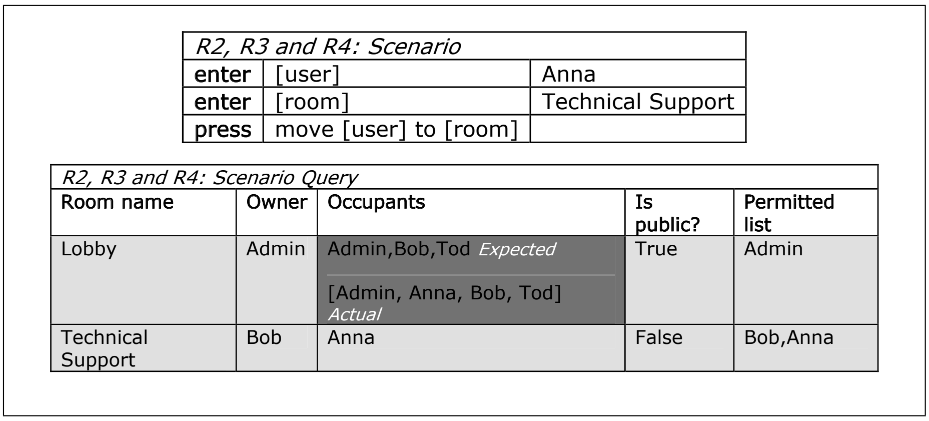
Fig. 15. Success: Result of executing tables in Fig. 13

Both tables in Fig. 15 succeed whereas the Row table in Fig 16 fails with a category 1 error indicating that Anna is in two locations at the same time (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 investigations of the code shows an implementation error in the body of routine `CHAT_SERVER.enter_room`. 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 (a 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. 17. The postcondition is:

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



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

Fig. 16. Failure: Result of executing tables in Fig 14

```
class CHAT_SERVER
        ...
        enter_room (u: CHAT_USER; r: STRING) is
                -- Move user 'u' into room with string name 'r'
            require
                $(u \neq Void) \wedge (r \neq Void) \wedge \neg(r.is\_empty)$
                $u \in location\_model.domain$
            ensure
                user_entered: $location\_model \cong (\textbf{old}\ location\_model \oplus (u \mapsto get\_room(r))$
                ownerships_not_changed: $ownership\_model \cong \textbf{old}\ ownership\_model$
            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:
    $(\forall\ r_1, r_2 \in own\_model.domain \mid r_1 \neq r_2 \bullet r_1.occupant\_model \cap r_2.occupant\_model \cong \varnothing)$
    coverage: $(\cup\, r \in own\_model.domain \bullet r.occupant\_model) \cong location\_model.domain$
end
```

Fig. 17. BON specification of the invariant for the `CHAT_SERVER`

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`.

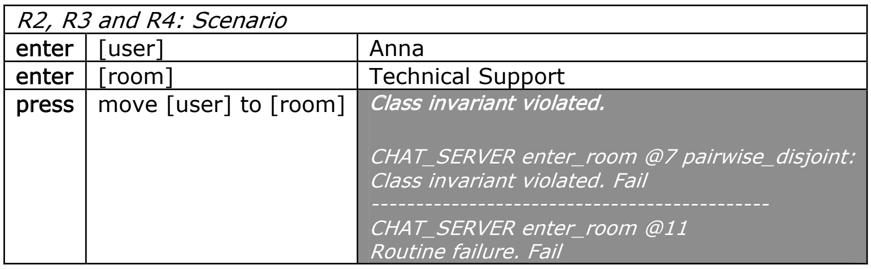
```
 1  class CHAT_SERVER
 2      ...
 3      location_model: ML_MAP [USER, ROOM]
 4      ownership_model: ML_MAP [ROOM, USER]
 5
 6      forall_rooms (r1: CHAT_ROOM): BOOLEAN is
 7          do
 8              Result := ownership_model.domain.for_all
 9                                              (agent empty_intersection (r1, ?))
10          end
11
12      empty_intersection (r1, r2: CHAT_ROOM): BOOLEAN is
13          do
14              if r1 /= r2 then
15                  Result := (r1.occupant_model ∗ r2.occupant_model) |=| create {
                        ML_SET[CHAT_USER]}.make
16              else
17                  Result := true
18              end
19          end
20
21      multi_union (s: ML_SEQ[CHAT_ROOM]): ML_SET[CHAT_USER] is
22          do
23              if s.count = 1 then
24                  Result := s.head.occupant_model.to_set
25              else
26                  Result := (multi_union (s.tail) |++ (s.head.occupant_model)).to_set
27              end
28          end
29
30  invariant
31    pairwise_disjoint: ownership_model.domain.for_all (agent forall_rooms (?))
32    coverage: multi_union (ownership_model.domain.to_seq) |=| location_model.domain
```

Fig. 18. Implementing the invariant using ML

The postcondition is correct but incomplete (as the error was category 1 and not category 2). As shown in Fig. 7, 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` as shown in Fig. 17.

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.

Fig. 18 shows how the invariants are written using the ML Eiffel library. 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. 18). 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



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. Specification violations are reflected to the Fit table

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 the Fit table 4.

As shown in Fig. 19 a category 1 error (expected vs. actual discrepancy) may indicate that the specification is either incomplete or even incorrect. A complete specification (via contracts) would have been flagged with a contract error in the Fit table. In the absence of a contractual specification error that pinpoints the faulty routine, there may either be an implementation error or specification error. By fixing the contracts we can pinpoint the precise routine that is not implemented correctly.

		Fit Table Violations
Specification Correctness	$P \wedge S \rightarrow R$	Actual vs. Expected (cat. 1)
Implementation Correctness	$C \rightarrow S$	Contract Violation (cat. 2)
System Correctness	$P \wedge C \rightarrow R$	

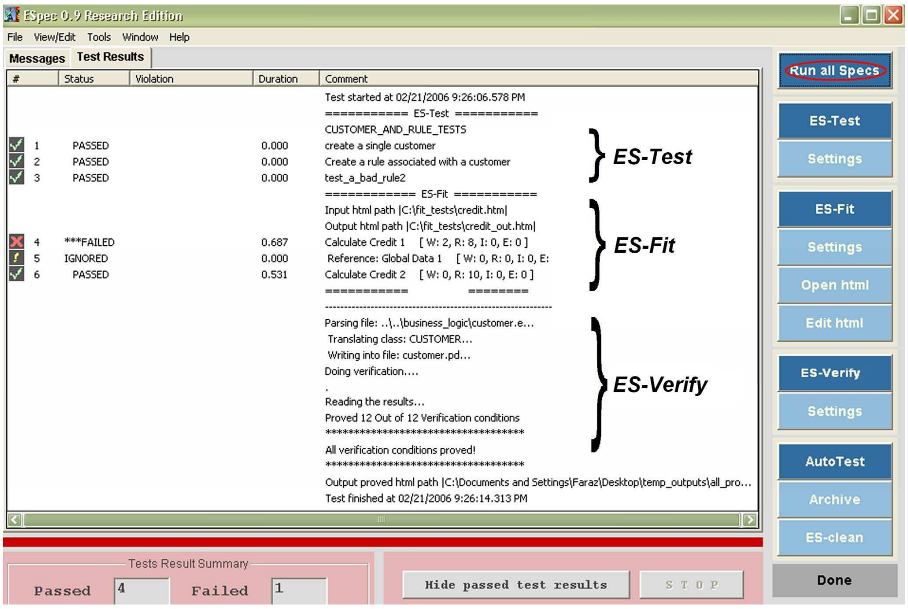
Fig. 19. Interpreting Fit table violations

6 ESpec tool

Fig. 20 shows the ESpec tool[1] in action. The tool allows the user to write testable customer requirements and design specifications that can be checked mechanically. ESpec provides feedback to the developer for Fit table requirement tests, Scenario Tests and unit tests as shown in the figure under a unified green bar

[1] `www.cse.yorku.ca/∼sel/espec`

(i.e. if all the various tests run correctly then a green bar is displayed). The tool also allows formal verification of implementations with respect to contracts using a theorem prover as described in [9].



```
#       Status      Violation      Duration   Comment
                                              Test started at 02/21/2006 9:26:06.578 PM
                                              =========== ES-Test ===========
                                              CUSTOMER_AND_RULE_TESTS
✓  1     PASSED                    0.000      create a single customer
✓  2     PASSED                    0.000      Create a rule associated with a customer
✓  3     PASSED                    0.000      test_a_bad_rule2
                                              ============ ES-Fit ===========
                                              Input html path |C:\fit_tests\credit.htm|
                                              Output html path |C:\fit_tests\credit_out.htm|
✗  4    ***FAILED                  0.687      Calculate Credit 1    [ W: 2, R: 8, I: 0, E: 0 ]
!  5    IGNORED                    0.000       Reference: Global Data 1    [ W: 0, R: 0, I: 0, E:
✓  6     PASSED                    0.531      Calculate Credit 2    [ W: 0, R: 10, I: 0, E: 0 ]
                                              ===========          ========
                                              ------------------------------------------------------
                                              Parsing file: ..\..\business_logic\customer.e...
                                               Translating class: CUSTOMER...
                                               Writing into file: customer.pd...
                                              Doing verification....
                                              .
                                              Reading the results...
                                              Proved 12 Out of 12 Verification conditions
                                              ***********************************
                                              All verification conditions proved!
                                              ***********************************
                                              Output proved html path |C:\Documents and Settings\Faraz\Desktop\temp_outputs\all_pro...
                                              Test finished at 02/21/2006 9:26:14.313 PM
```

Tests Result Summary

Passed 4   Failed 1

Fig. 20. The Espec software quality workbench

References

1. Scott Ambler. Agile Model Driven Development is Good Enough. *IEEE Software*, 20(5):71–73, 2003. Agile Model Driven Development is Good Enough.
2. Tatiana Andronache. The english language as an effective it tool. *Computerworld*, page 18, Feb. 2007.
3. Mike Barnett, Rob 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.
4. 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.
5. Standish Group. Project management: The criteria for success. Software Magazine, February 2001.
6. C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

7. Michael Jackson. *Problem frames: analyzing and structuring software development problems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
8. Rick Mugridge and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests.* Prentice-Hall, 2005.
9. Jonathan Ostroff, Chen-Wei Wang, Eric Kerfoot, and Faraz Ahmadi Torshizi. Automated model-based verification of object oriented code. In *Verified Theories: Theories, Tools, Experiments (VSTTE Workshop, Floc 2006).* Microsoft Research MSR-TR-2006-117, 2006.
10. Jonathan S. Ostroff and Richard F. Paige. The Logic of Software Design. *Proc. IEE - Software*, 147(3):72–80, 2000. The Logic of Software Design.
11. 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), Sep-Oct 2005.
12. Richard Paige and Jonathan S. Ostroff. The Single Model Principle. *Journal of Object Oriented Technology*, 1(5), 2002.
13. 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.
14. Faraz Ahmadi Torshizi and Jonathan S. Ostroff. ESpec – a Tool for Agile Development via Early Testable Specifications. Technical Report CS-2006-04, York University, Toronto, 2006.