



The United Nations
University

UNU-IIST

International Institute for
Software Technology

Specification for Testing

**Chris George, Padmanabhan Krishnan,
P A P Salas and J W Sanders**

June 2007

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit our home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director



The United Nations
University

UNU-IIST

International Institute for
Software Technology

P.O. Box 3058
Macao

Specification for Testing

**Chris George, Padmanabhan Krishnan,
P A P Salas and J W Sanders**

Abstract

The success of model-based testing, in automating the testing of an implementation given its state-based (or model-based) specification, raises the question of how best the specification can be tweaked in order to facilitate that process. This paper discusses several answers. Motivated by an example from web-based systems, and taking account of the restriction imposed by the testing interface, it considers both functional and non-functional properties. The former include laws, implicit system invariants and other consistency conditions whilst the latter include security leaks. It concludes that because of the importance of the link between specification and implementation in the testing process, there is a trade-off between genuinely useful testing information and the incorporation of some degree of information about the link, not normally regarded as part of the specification.

Chris George is Associate Director of UNU-IIST. He is one of the main contributors to RAISE, particularly the RAISE method, and that remains his main research interest. Before coming to UNU-IIST he worked for companies in the UK and Denmark.

Paddy Krishnan is Professor of Computer Science in the School of Information Technology at Bond University, Australia, and a Research Associate of UNU-IIST. His interests lie in program specification and verification.

Percy Salas is a KJRoss PhD student at Bond University, supervised by Paddy Krishnan in model-based testing. He has been a fellow at UNU-IIST and a lecturer at the Universidad San Pablo in Arequipa, Peru.

Jeff Sanders is Senior Research Fellow at UNU-IIST, having recently joined from the Programming Research Group at Oxford. His interests lie largely in Formal Methods.

Contents

1	Introduction	1
1.1	Related Work and Outline	2
2	Case Study	2
2.1	Overview	3
2.2	Model: Webbo	3
3	Linking Model to Implementation	7
3.1	Testing Language	8
3.2	Action Words	9
3.3	Oracle	9
3.4	Link Issues for Testing	10
3.5	Problems with Test Automation	10
4	Laws	11
4.1	Laws in Webbo	11
4.2	Reality Check	12
5	Testing Consistency	13
5.1	Invariants	13
5.2	Confidence Conditions	14
5.3	Confidence Conditions in Webbo	15
5.4	Partial Operators	16
5.5	Are Equalities Assignments?	16
5.6	Guide to Implementers	16
6	Boundaries	17
7	Intermediate States	18
7.1	Sequential Components	18
7.2	Principle	19
8	Conclusion and Further Work	19

1 Introduction

(Formal) specifications and implementations are normally viewed as being poles apart. After all, a specification captures requirements by expressing what a system should achieve, whilst the purpose of an implementation is to be executed and so it contains much detail whose concern is computational efficiency; the connection is of course that the implementation conforms to the specification—ideally! They might also be viewed as being poles apart because, after all, much of the process of system development lies between a specification and an implementation.

Because that conformance *is* only ideal, testing is required. All testing presupposes *a priori*, an oracle, knowledge for interpreting test outcomes: of which tests pass and which fail. That knowledge constitutes, for any system of realistic size, only partial information about the specification. But it demonstrates an unbreakable link between specification and testing.

The advent of the important area of model-based testing, MBT, [7, 9, 26] forges an even stronger link. Its primary importance is the complete automation of validation testing, subject to control by the test engineer of those features of the system being tested, of coverage criteria and so on. As pointed out by Utting [25], MBT relies on *redundancy* between the test specification and the implementation; and then it is equally likely to reveal errors in each.

Over the past couple of decades, considerable experience has been gained in specification. The cost of the time spent on specification during the critical early stages of the development cycle can be partly amortized over the later activity of testing using MBT. However the precise difference between a (standard) specification and one that serves as a model for MBT is still unclear. That is the topic of this paper. More specifically, its purpose is to consider the following question, particularly in the context of web-based systems:

What can a specifier do, when constructing a (state-based, or model-based) specification, to facilitate the subsequent task of testing, particularly by automated techniques like MBT?

The nature of testing has changed as a result of specification. One of the benefits of specification is that it provides notation and a place in the development cycle to make precise conditions that once were routinely incorrect in code; then they were picked up only by testing. Perhaps the best examples of that nature are boundary conditions: indices out of bounds, loops iterating too many or too few times, *etc.* Of course the specifier might still get those details wrong, but with MBT the mistake is likely to be picked up in the model rather than in the implementation.

One of the strengths of MBT is that the model and tests generated from it can be reused even when the implementation changes. This means that MBT is useful in blackbox testing, whose focus is on the properties of the system rather than the low-level coding details. This facilitates the development of a model that is abstract and is derived only from observable behaviour. This is not to say that MBT cannot be used in whitebox testing. But the more the details of the implementation are exposed, the less abstract the model—and hence the less reusable—it becomes.

In this paper, however, it is argued that some implementation details have to be exposed for certain types of property. The properties addressed include standard ‘functional’ properties like laws between operations, system invariants, confidence conditions (like pre- and postcondition analysis), boundary analysis; and they include ‘nonfunctional’ properties like security leaks.

For MBT to be really useful, the tests generated from the model must be linked to the implementation. The link must provide sufficient information to automate the process of testing the implementation from the model. It acts as an ‘action refinement’, translating test sequences obtained from the model into suitable test sequences for the implementation. The *action word* approach [6, 5] has been proposed as a simple way to establish the link. This approach applies naturally to models which are labelled transition systems whose labels represent aspects of functionality that are to be tested. The link then associates code, which actually performs the testing, to the label. For example, if the link associates the code *ca* to the label *a* and *cb* to the label *b*, then the test sequence *ab* results in execution of the code *ca ; cb*. Usually the oracle is built into the code.

1.1 Related Work and Outline

The study of the interplay between specifications and testing is far from new. Of the many contributions which predate MBT, a large number can with hindsight be seen as precursors. *Specification-based testing* is a term that has for long been used to describe the ‘generation’ of tests from a (formal) specification.

Some work has focused on the animation of specifications, either by choice of specification notation [7, 13, 14, 15, 18] or by translating the specification into executable form [11, 12, 19]. Incorporation of the notion of testing into the framework of Formal Methods, and the refinement calculus in particular, has been accomplished by Aichernig [1]. Testing the kind of system considered here, specifically to reactive systems, has been the subject of a Dagstuhl meeting [4].

In outline, the paper proceeds as follows. In Section 2 a case study is presented of a web-based network-management system, and in Section 3 the manner in which it is linked to an implementation is discussed. Then three kinds of functional property are considered for the generation of extra-specification tests, in Sections 4, 5 and 6. In Section 7 a non-functional property, security, is considered. The case study is used illustratively throughout.

2 Case Study

This section describes a web-based network-management system that helps to illustrate the key issues discussed in this paper. The system is based on a real web application, whose testability has been determined by the interface available.

First, a brief informal overview of the system behaviour is given (Section 2.1) and supplemented (in Section 2.2) with a formalisation in ObjectZ. Then follows (in Section 3) a discussion of those issues in

the model that are relevant to testing; that is followed (in Section 3.5) by a brief discussion of problems related to testability.

2.1 Overview

The system consists of a simple web-based system for managing a network of machines remotely. It is based on a network of clients and their users. There are two special kinds of user: administrators and managers. Although both are users, no user is both an administrator and a manager.

The system requires users to authenticate themselves by logging on in the usual manner. For security purposes, three consecutive unsuccessful logins result in the user's account being locked; this level of abstraction overlooks the details of how that is undone by a manager or administrator). A user can log on to only one client at a time.

An administrator (and only an administrator) can create a client, provided it is not already present and provided that a manager is assigned to it. An administrator can remove a client, provided there are no users logged on there. An administrator can select a client and see its details, consisting of the client's manager and the users currently logged on there. An administrator can also see the details of a user, consisting of its password, whether it is logged on and if so where, how many consecutive unsuccessful attempts it has currently made, and email it has sent and received. Finally, an administrator can broadcast mail to all users, and can send mail to a specific user as if from another specific user.

A manager (and only a manager) can create a user account, provided the user is not already registered. A manager can remove a user, provided it is not logged on, is not itself a manager or administrator, and provided mail it has received is also removed. After an administrator or manager logs out the functionality associated with its roles cannot be invoked.

Other users have very limited capabilities (at this level of abstraction). They can log in on any client, read their messages and send messages to other users and log out. But they cannot manage users or clients.

Initially there is at least one administrator and at least one manager (just because this level of abstraction abstracts the actions that change those sets), there are no other users, no user is logged in nor has tried to log in, and there is no post. It is arbitrary where the administrators and managers are initially logged in.

2.2 Model: Webbo

The model of that web-based network-management system is called Webbo. The specification below is approximately ObjectZ [8] but uses dependent types in order to shorten schema invariants and uses $C \sqcup D$ to stand for the disjoint union of C and D (*i.e.* the union in which the invariant $C \cap D = \{\}$ holds).

These generic types are assumed: *Users* for the set of possible users; *Clients* for the set of possible clients; *Mess* for the set of email messages; and *Pass* for the set of all possible passwords, both valid and

invalid (thus *Pass* might well consist of the set of all character strings constrained in some way by length; but such detail is ignored at this level of abstraction).

These schemas are required and defined in the specification. *Loginfo* provides for each user: the number *no* of putative logins in the current sequence of attempts by that user; the user's password *pd*; a Boolean *in* which is true iff the user is currently logged in; the client *at* at which, if *in* is true, the user is logged in; and a Boolean *lock* which is true iff the user's account is locked (thus *lock* is a redundant observable, equivalent to $(no = 3)$).

State is described by sets: *U* of users, *A* of system administrators, *M* of managers of clients, *C* of clients; by a function *mng* which assigns a manager to each client (so is a total function), a function *log* which assigns *Loginfo* to each user, and a relation *post* between a sender-receiver pair and messages that is in general many-to-many. See Figure 1.

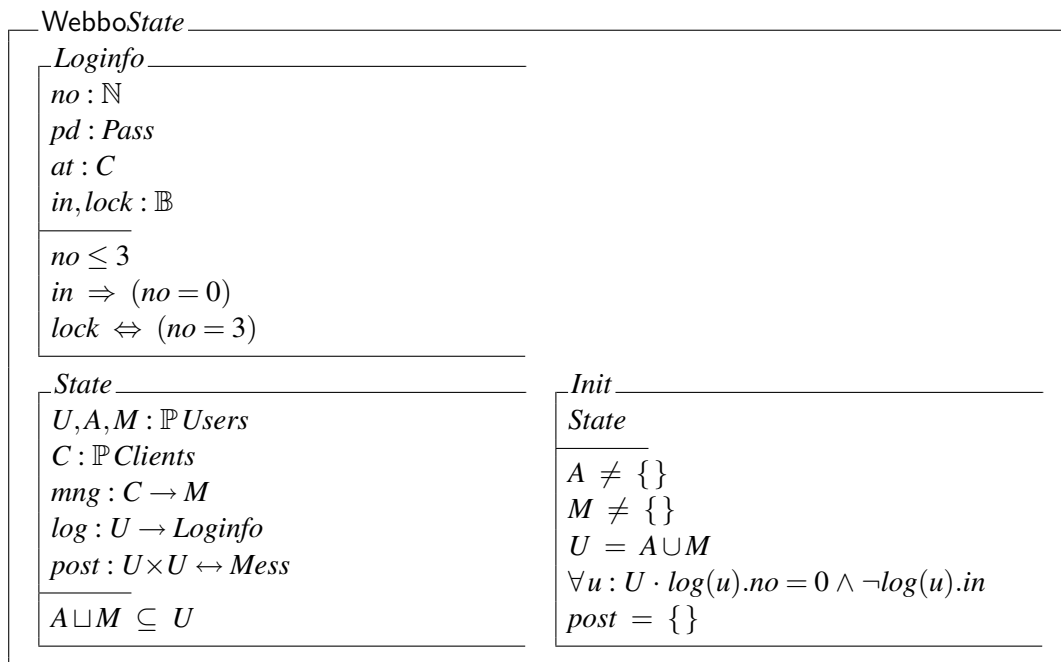


Figure 1: State for the model Webbo.

An administrator can create a client, with operation *CreateClient*, only if logged in, the client is not already in the network, and by assigning a manager to the client. Removal of a client, by operation *RemoveClient*, is similar, except that for a client to be removed it must have no users logged in there. See Figure 2.

With operation *AdminClient*, an administrator *a?* can receive information about the manager *man!* of a client *c?* and about which users, *users!*, are logged in there. With operation *AdminUser* an administrator can receive information about a user *u?*. See Figure 3.

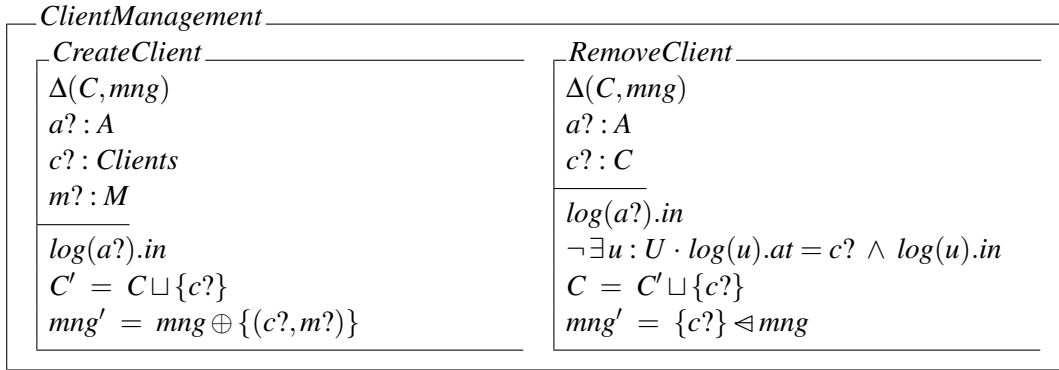


Figure 2: Client management in Webbo.

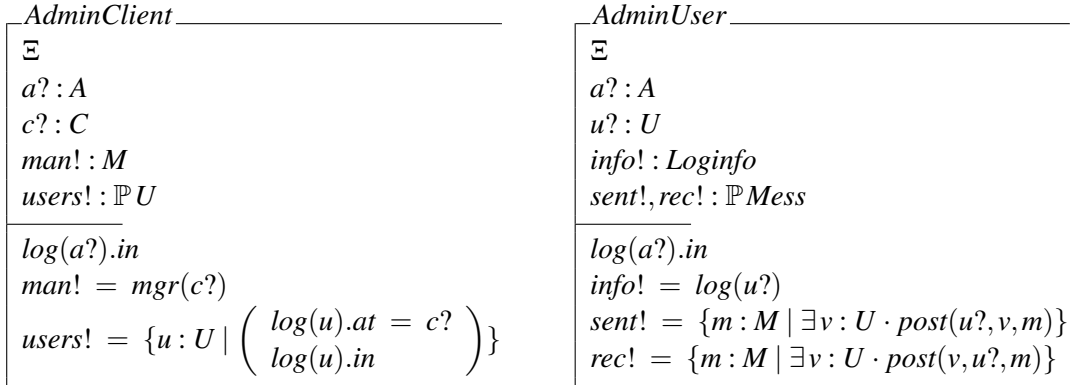


Figure 3: Administrative information in Webbo.

An administrator $a?$ sends post either by broadcasting a message $p?$ to all users from itself, with operation *AdminPostAll*, or by sending it to just a single recipient $r?$ from user $u?$, with operation *AdminPostOne*. See Figure 4.

The ‘management’ of users extends that of clients, because when a user is removed by a manager, its email is also removed; the user must not be logged in, and must not be an administrator or manager. This description abstracts the initial choice of a client’s password. See Figure 5.

A user $u?$ who is logged in may send a message, with operation *UserPost*, to a named user $r?$; or may read mail without deleting it (an easy modification updates the state to remove read post), by outputting the mail in the set $ms!$ of messages. Reading mail does not change the system state; sending it updates only the state component *post*. See Figure 6.

But the most subtle action is that of logging in. A user may (attempt to) log in only if not already logged in. The attempt fails if the input password $id?$ is wrong; on the third consecutive failed attempt the user

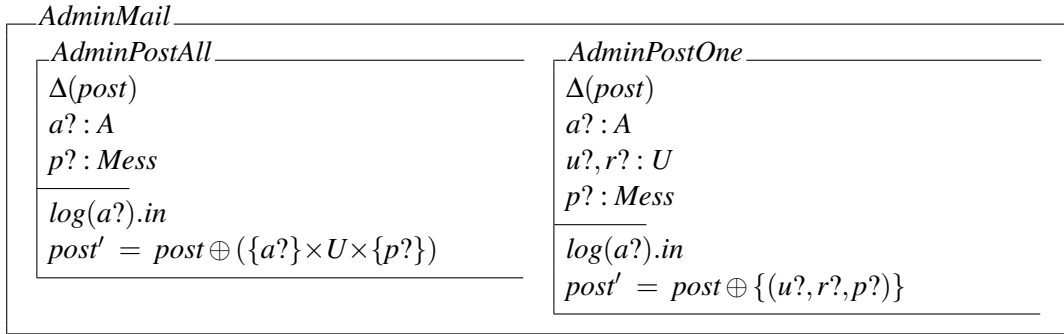


Figure 4: Administrative posting in Webbo.

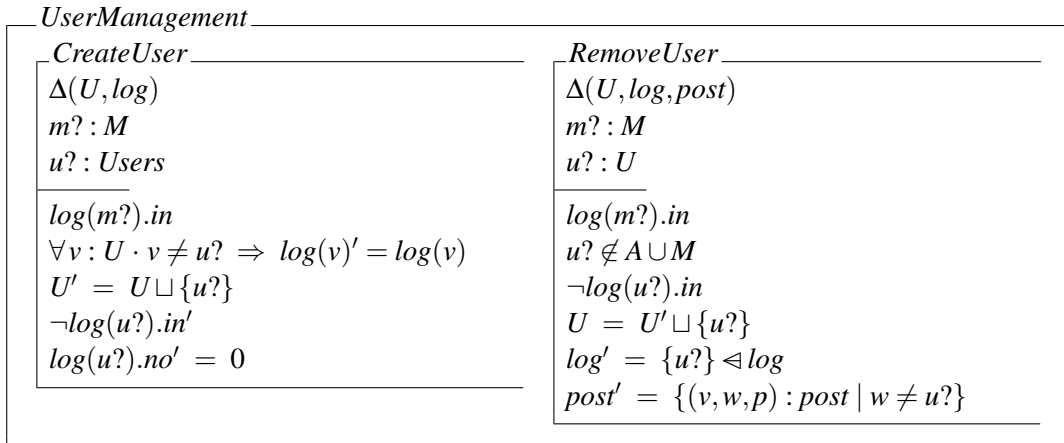


Figure 5: User management in Webbo.

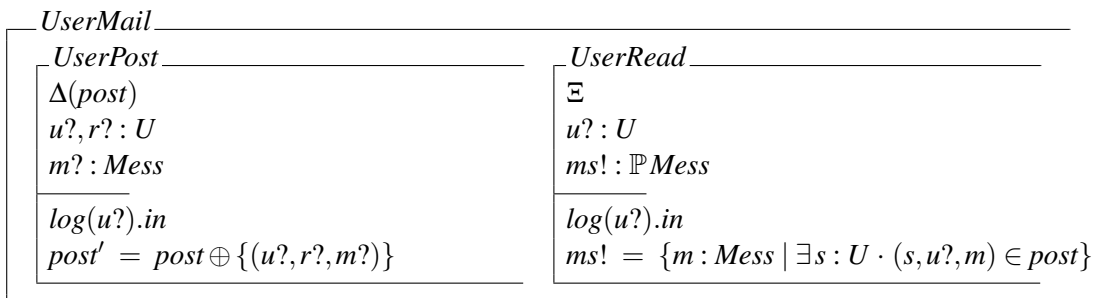


Figure 6: Sending and reading post in Webbo.

is locked out, but otherwise another attempt is permitted. A successful attempt returns the count *no* to 0

and changes the status of the user to being logged in. A successful login returns an *ok* acknowledgement whilst an unsuccessful one returns *fail*. Operation *Login* is described in Figure 7.

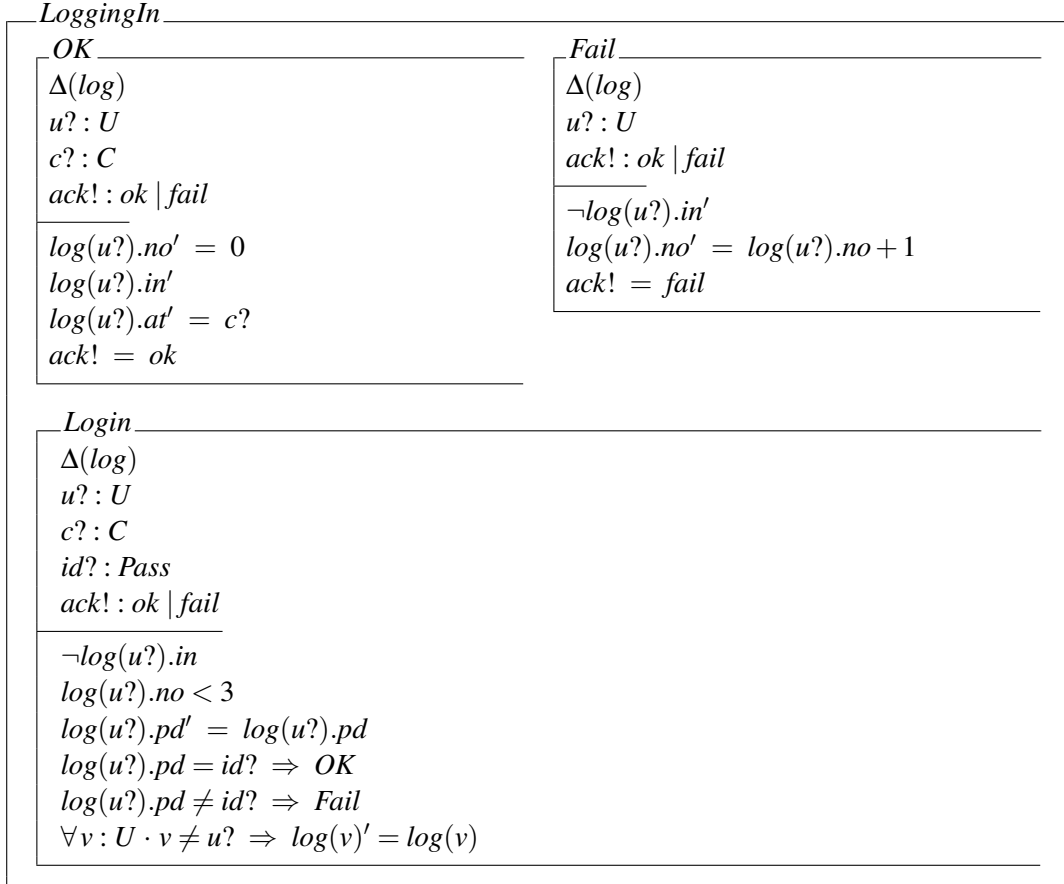


Figure 7: Logging on in Webbo.

Logging out is straightforward and is described in Figure 8. User $u?$ is logged in before the operation but not after it; its password remains unchanged as does its number of unsuccessful login attempts (at 0); however $log(u?).at$ is left arbitrary.

Overall Webbo is specified with a class that combines the previous descriptions; see Figure 9.

3 Linking Model to Implementation

This section outlines how the link between the model and the implementation can be developed for the purposes of testing. Several examples are presented to illustrate the main issues, since a complete description lies beyond the scope of this paper.

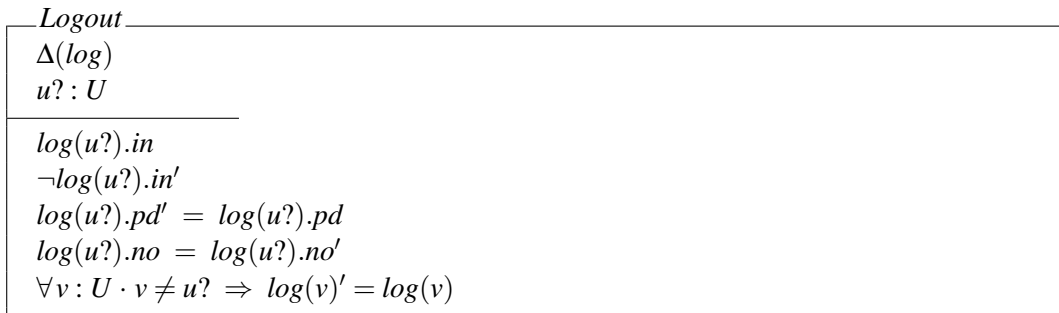


Figure 8: Logging out in Webbo.



Figure 9: The overall system Webbo.

3.1 Testing Language

General purpose programming languages are in general used to test systems (*e.g.*, Java is used in the Korat framework [3] and Ruby/Watir is used for testing web based systems [16]). This ensures that the expressive power of the tester is not restricted. But the implementation must provide sufficient information to the tester otherwise the expressive power of the testing language cannot be utilised.

For example, the Ruby/Watir framework uses the Internet Explorer's Component Object Model (COM) interface to access objects of interest. However if the HTML does not name the objects it is often difficult to pick the right object [16] and verify its state. In object-oriented approaches information hiding can restrict the ability of a tester to control and observe the state of the computation [2]. The implementation must provide an interface which can be used to inspect the values of relevant variables in the testing process.

In the present paper no particular testing language is assumed, though the constraints just mentioned will play a deciding role.

3.2 Action Words

The approach taken here to linking the specification and implementation is that of ‘action words’ due to Buwalda and Kasdorp [5, 6]. It may be described, in the current context, as follows.

The atomic input action $pd? : Pass$ is interpreted in the implementation as a succession of keyboard inputs, via a web page, as defined by a finite automaton. The automaton ensures that the password has length within an allowable range and so is legal.

Whilst the abstract event is atomic its concrete translation is not, since it now has several states (to reflect the length of legal passwords) and the possibility of being interrupted. Nonetheless the result is a bijective representation of *Pass*, that forms the link which is animated by code.

Before providing examples, the appropriate notion of oracle must be defined.

3.3 Oracle

An operation applied inside its precondition and producing a result within its postcondition is thought of as forming a ‘positive’ transition whilst one applied outside its precondition or producing a result outside its postcondition is ‘negative’. Extending that idea to sequences of operations defines the oracle, which provides the required notion of a test that passes (all components in the trace are positive) and one that fails (at least one is negative).

For example, the (positive) transition $\text{login}(u, up)$ indicates that one can log in as u with password up . For this, the initial web page must be at starting page and the result must be the page that is displayed after a successful login. As indicated above, the link between the model and the implementation translates the user name u and putative password up to appropriate text fields and the process of logging in involves clicking the button. It also links input of up to a succession of keystrokes (and perhaps to an acknowledging output, audible and visible). For this link to be useful in test automation, methods to access the text fields and buttons are assumed to be made available by the COM structure.

The model also contains (negative) transitions of the form $\text{loginFail}(i, p)$ which indicate that the user i with putative password p fails the login. While the link for the process of logging in is the same as for login , the linking of the results is quite different. When login succeeds, the web page has the menu item with the appropriate functionality; whilst if the login fails, the web page does not have the menu item but goes back to the retry page (that is, until the user is locked out). This again assumes availability of the appropriate methods to inspect the resulting web page.

3.4 Link Issues for Testing

The model exhibits the behaviour that three consecutive `loginFail(i,_)` result in locking the user's account. To build the oracle into the action word, one might use `loginLock(i,p)`. Again the link has to translate the login process as before, but with a different check on the outcome.

The implementation may have state variables such as Booleans `auth` and `lock` which may not be directly accessible from the web page. In this particular case the implementation generates output on the web page by setting the title to "Not Logged in" or "Logged in".

If the link has only a method to examine the title, it is not possible to determine if the account is locked. That is, both `loginFail(i,p)` and `loginLock(i,p)` may be forced by the link into checking the title to "Not Logged in".

Either the title has to change or a method to examine `locked` is essential. If there were a message displaying 'locked' then the web page can be tested by checking its contents.

Sometimes there are popups (say created by Javascript) that require confirmation. In the model there are transitions of the form `clickOK`. They need to be translated into a number of operations—getting a handle of the javascript window, clicking the okay button there, waiting for that window to disappear and the main window to go to the desired page.

Usually all, or anyway most, of the methods are available or can be made available but if the web page does not provide that information, testing fails.

3.5 Problems with Test Automation

The purpose of this section is to show that developing a model and a link from the model to the implementation cannot automate all aspects of testing. The standard black box testing approach to web systems is thus not sufficient.

The first problem arises since an administrator can send, with operation *AdminPostOne*, a message on behalf of a user. The precondition for this operation is that the user posting the message should be an administrator and the message should be addressed to an existing user from an existing user. The model also defines the expected result of the operation: the message should be added to the pool of messages (various designs for which are discussed in Section 5.6).

Assume that an administrator *a* posts a message on behalf of the user *u* to *r*. After posting the message the system returns *a* to the web page from where the messages can be managed. Administrator *a* has no direct way of verifying that the message has been added to the post. That is, it is not possible to define a link using the methods made available by the web system.

The link needs to include steps such as logging out from administrator *a*'s account, logging in to user *u*'s

account, opening the post and then viewing the message. Those steps assume that the tester (or the test execution engine) knows the password of r 's account. If this assumption is false, the situation cannot be tested. If the password is not known, the test case for the *CreateMessage* function can define a series of preparatory steps that includes the creation of a specific dummy user for whom the tester should record the password to be used later. Administrator a then creates a message for this dummy user. However, the testing is restricted to just that dummy user.

A similar situation exists while creating a client. The implementation does not provide any actual information about the status of the operation. There is no way way of verifying that the client has indeed been created and users for it can be created. The link must try to add a user to the client and then delete the user. If this succeeds the client has been created. In this case the person creating the client is an administrator and does not have the right to create a user. Hence the test script needs to logout and login as manager to test the creation of a client. The system must allow this process for the testing to be automated.

Such situations are now examined in a more systematic fashion and it is shown how these problems lead to a tradeoff between specification and implementation.

4 Laws

State-based specifications and algebraic specifications feature quite different styles of requirements capture. Nonetheless the state-based specifier is typically aware of laws relating the operations being described in the state-based notation. For example, a 'do' operation is often accompanied in a system by an 'undo', in which case their composition—perhaps under some assumption, or by abstracting some components of state—leaves state unchanged. The following examples demonstrate those ideas using Webbo.

4.1 Laws in Webbo

Creation of a client followed by its removal leaves the state of Webbo unchanged. Conversely, removal of a client followed by its creation may not leave state invariant because the manager assigned during client creation may not coincide with that just deleted in client removal; but other state components are unchanged.

For this some notation is required. Suppose that (state) schema S includes an observable $a : A$. Then the schema with that observable abstracted, and its identity function, are defined:

$$\begin{aligned} State_a &= \exists a : A \cdot State \\ \text{id}[State_a] &= \lambda s : State_a \cdot s. \end{aligned}$$

(In practice it may be simpler to list, as subscripts there, the state components that are present rather than those that are not; but this notation is more convenient in theory.)

Now the previous laws can be expressed:

$$\begin{aligned} \text{CreateClient} \circledast \text{RemoveClient} &= \text{id} \\ \text{RemoveClient} \circledast \text{CreateClient} &= \text{id}[\text{WebboState}_{\text{mng}}]. \end{aligned}$$

Creation followed immediately by removal of a user does not change the system state. But conversely, removal followed by creation of a user may result in a different password $\text{log}(u?).pd$ and a different $\text{log}(u?).at$. Continuing the subscript convention, that is expressed

$$\begin{aligned} \text{CreateUser} \circledast \text{RemoveUser} &= \text{id} \\ \text{RemoveUser} \circledast \text{CreateUser} &= \text{id}[\text{WebboState}_{\text{pd,at}}]. \end{aligned}$$

Further laws in Webbo follow from the following observations. The action of a user posting mail to another does not have the same effect as that of an administrator performing an analogous *AdminPostOne*. For it is a precondition of the former but not the latter that sender $u?$ be logged in, and a precondition of the latter but not the former that administrator $a?$ be logged in; but otherwise the effects are the same.

From a fresh start, $\text{log}(u?).no = 0$, three unsuccessful logins of user $u?$ leave the system in a state in which all components are the same except that $\text{log}(u?).no = 3$, $\text{log}(u?).lock$ holds and $\text{log}(u?).at$ is arbitrary.

A successful login of a user followed by its logout thus leaves state unchanged except for the $\text{log}(u?).at$ component. However a logout followed immediately by a login of the same user leaves state unchanged only if $u?$ was originally logged in and at the same client as the login.

All provide opportunities, subject to the qualification above about implementation detail, for testing; and all suggest information that can be provided by the specifier.

4.2 Reality Check

However in real systems the check for identity requires the checking of all system components. For example, it requires checking that the database has all and only the information that was present at the start of the operations. Expense precludes this from being part of the testing process. But worse, such simple identities are almost never true. Databases and other applications (such as the authentication engine) keep log files for rollback and forensics. As these log files do not affect the functionality they rarely feature in specifications.

Thus for testing purposes it is essential to specify explicitly the components that are used (or not used) for the calculation of the identity relation; hence our use of the notation above. In practice that requires a combination of knowledge of both specification (to determine which components to observe) and link (how they are represented in the implementation).

5 Testing Consistency

Different (state-based) specification notations employ slightly different styles and hence place emphasis on slightly different concepts. A state invariant in Z (like the three explicit predicates that constrain *Loginfo*¹ or the single explicit predicate that constrains *State*) constitutes a sub-type in RAISE. Because of their importance in testing, both approaches are considered here.

5.1 Invariants

System invariants can be thought of as generalising algebraic laws. They are strictly more general: a property may hold after an operation is invoked even though it may not be captured in a law. As has just been seen, Webbo contains both explicit and implicit invariants. A further type of implicit invariant holds not from the expansion of a dependent type but from a property that holds initially and is maintained by all operations. An example is the nonemptiness of *A*. In principle each invariant provides an important test (if it can be achieved in the web-based implementation). This is something the specifier is in a position to provide, and will typically have considered either making explicit in the specification, or proving is a consequence of it.

An example of a restriction on testing imposed by the web-based interface is in testing the (explicit) invariant

$$\forall u : U \cdot \log(u).no \leq 3.$$

That cannot be tested directly without access to the internal variable that records the number of consecutive failed logins (something the web-based interface does not allow). For instance, in most applications we cannot directly test if *log(u).no* has the value 2 after two unsuccessful logins. Thus the invariant must be changed, to incorporate

$$\log(u).lock \Rightarrow \text{LoginFail}(u, _).$$

This can be tested at the state where the lock is first set (when *log(u).no* equals 3). At this point any attempt to invoke *Login* on that user account should fail. The fact that the account is locked can be obtained in most implementations.

Another kind of invariant is that subtypes are used in a consistent manner: if an operation with inputs or states in a subtype is supposed to preserve that subtype (in the sense that outputs or state after also lie in the subtype), then it actually does so.

Less comprehensive invariants, nonetheless important for testing, are typically conveniently expressed in temporal logic. It has been shown [17] that in some circumstances such temporal formulae can replace the model-based specification for the purposes of testing. But in general just as a state-based specification can be complemented by algebraic laws—or refinements, *i.e.* one-sided laws—so too it can be complemented by temporal properties that the specifier is in a position to posit.

¹The statement $at : C$ provides an implicit constraint since it expands to the type statement $at : Client$ with explicit constraint $at \in C$.

5.2 Confidence Conditions

*Confidence conditions*² are certain kinds of condition on a specification whose failure, it has been found by experience, often indicate error. They need be neither necessary nor sufficient for consistency³, but their truth provides a degree of confidence in the specification.

Confidence conditions therefore provide some opportunities for verification activities. The RAISE tools, for example, allow them to be generated and inspected, to be generated and proved (in PVS), to be model checked (in SAL), and to be included in executable code produced by translators. Confidence conditions are used in two ways: to look for inconsistencies in the specification and also as hints for possible extra code in the implementation, to be switched on during testing and possibly also to aid in debugging. In this paper it is the second usage that is of particular interest.

In (sequential) RSL the main confidence conditions are:

- subtypes are not empty
- arguments to functions are in subtypes
- results of functions are in subtypes
- assignments to variables are in subtypes
- preconditions of functions are satisfied by invocations
- postconditions of functions are satisfied by invocations
- case expressions are complete.

Here ‘functions’ include built-in functions and operators, like division (perhaps by zero) or taking the head of a list (perhaps empty).

Although those conditions might be expressed slightly differently in Z (schemas are satisfiable, and functions and expressions are well defined at both their points of definition and of use) they are equally important. The same ideas can be applied to any specification language, but the possibilities will change with the language. In general, there is a tendency for Z specifications to involve less redundancy than RSL ones, and so the possibility for generating confidence conditions may be reduced. The next section applies the idea of confidence conditions to Webbo.

²The term was coined in the RAISE project [20, 21].

³Lack of sufficiency is obvious. They need not be necessary because of limitations in the static analysis that generates them; for example the (low-level, but consistent) expression ‘**if false then 1/0 else 1 fi**’ might generate the confidence condition that the divisor 0 is not 0, which fails to hold.

5.3 Confidence Conditions in Webbo

Loginfo, with its three-predicate invariant, forms an RSL subtype. The *State* definition has an invariant consisting of one explicit and one implicit predicate (the implicit predicate $mng : C \rightarrow M$ expands to the type statement $mng : Clients \leftrightarrow M$ with an explicit constraint $\text{dom}(mng) = C$), and so also forms an RSL subtype. Thus obvious checks are

- Operations generate states consistent with the *State* predicate (which includes the *Loginfo* predicates on each value in the range of *log*)
- Outputs of type *Loginfo* satisfy the *Loginfo* predicates.

The specification Webbo is written in a largely implicit style. Thus in *RemoveClient*, the condition $C = C' \sqcup \{c?\}$ defines C' implicitly whereas the equivalent $C' = C \setminus \{c?\}$ does so explicitly (but by violating the symmetry with *CreateClient*). Implicit descriptions are typically more abstract and aimed at establishing properties (for which purpose symmetry is helpful), whilst explicit descriptions are typically closer to code; most specification notations permit both styles, for use as appropriate in the development cycle.

Since an implementation models all the predicates in its specification, it is difficult to see, of an implicit-style specification, where possible inconsistencies come from: if a function is defined by a postcondition the confidence condition becomes an assertion that a result satisfying the postcondition, plus any relevant subtype condition, exists. This does not tend to be very helpful in discovering errors. But when an explicit style is used then a test can be run to check that a result lies in the appropriate subtype.

In a notation (like RSL) whose explicit functions may also have postconditions, the redundancy permits a much stronger confidence condition to be generated. Similarly with explicit preconditions (like RSL, VDM or the refinement calculus), which permit a check that the explicit precondition is strong enough to ensure the preconditions of functions and operators used in the definition. (The calculation of preconditions in Z provides no redundancy, although it may introduce mistakes in the specification—an inconsistency⁴ between the calculated precondition and the implicit one—to be picked up in testing [25].)

So it is seen that in Webbo the subtype checks are not so useful as checks on the specification, although they do provide checks to be included in the implementation. The checks will probably need to be optional, as some will be computationally expensive, but they can be used in testing and debugging.

⁴Typically because the specifier does not do the calculation carefully enough, relying on intuition for the answer.

5.4 Partial Operators

The appearance of disjoint union, \sqcup , in Webbo exhibits the use that can be made of partiality. It appears in several operations in the form

$$S' = S \sqcup T$$

from which follows the precondition that S and T are disjoint. It also appears in the form

$$S = S' \sqcup T$$

from which follows the postcondition that S' and T are disjoint, though that is subsumed in the more general postcondition which is the whole equality.

5.5 Are Equalities Assignments?

When a specifier in ObjectZ writes

$$e0 = e1$$

where $e0$ and $e1$ are general expressions, then the confidence condition that one might generate is that there is a non-empty intersection between the possible sets of values for $e0$ and $e1$. But when the left-hand side of the equality is $x!$ or x' , where x is a variable whose type is a subtype, the equality can be regarded as an explicit specification of an assignment in the implementation, in which case the stronger check can be introduced that the value of the expression on the right-hand side lies within the variable's subtype.

5.6 Guide to Implementers

A number of checks can therefore be generated to apply as optional code in the final implementation. However their execution requires:

- Equality functions on all types are needed. Since, as mentioned elsewhere, the finally implemented types are typically richer, an abstract equality needs to be implemented which is obviously defined in terms of the (abstract) equalities on the components
- In general, the constituents of the specification need to be implemented, or some means found to express the conditions in terms of the implementation.

For example, the relation *post* is unlikely to be held as a set: it will be calculable (in theory) as such a relation from, perhaps, individual mail boxes for recipients like one which associates to each receiver a set consisting of pairs comprising a sender and a message

$$postbox : U \rightarrow \mathbb{P}(U \times Mess)$$

subject to the coupling invariant

$$post = \{(u, r, m) : U \times U \times Mess \mid u \in U \wedge (r, m) \in postbox(u)\}.$$

Then the assertion that a message exists with someone as recipient is easy to implement. The assertion that some message exists with a given sender would be harder (unless the implementation followed a design in which each user kept an analogous *outbox* of sent post).

- Following from the previous point, it is clear that the code used to implement these checks will often break properties such as security that need to be imposed on user-accessible functions. How to deal with this is unclear. For example one user ought not to be able to access another's post (except perhaps the sender, if the *outbox* design is used). Yet to test the operations of *AdminPost* and *UserMail*, just such probing is necessary. However this treatment has stopped short of security and other such properties.

6 Boundaries

Boundary value testing is an important technique that has been adopted in MBT [24]: each operation is tested at each state that is (reachable and) extreme in a sense defined by the tester. In *Webbo*, boundary testing focuses on the extreme values of the *Loginfo* observable $no = 0, 3$, on both values of any Boolean variable, on the initial values of the set *State* observables, on the empty and full cases of *post*, and so on.

The specifier is in a position to define appropriate boundary functions, which the tester might not think of. For that purpose it is sometimes easier (compared with the method used in [24]) to define an order (like set inclusion or ordering on integers) and to identify a boundary with its extreme points. That yields all the examples mentioned above.

One interesting, less systematic, example already considered is identification of a dummy user to include as a boundary point, as discussed in Section 3.5. But this is information a specifier has only if familiar with the test constraints.

Another example is provided by the important operation of logging in. There are potentially infinitely many ways in which the login operation can yield *Fail*. Of course in testing, a suitable subset of inputs must be found to trigger that behaviour. However, the model does not explicitly state which inputs to consider. Again, it is appropriate for the specifier to augment the specification with information to aid testing. For instance, a suitable function f can be defined with the property that if $Login(u?, id?)$ succeeds then $Login(u?, f(id?))$ should fail.

Similarly, tests $Login(ux, idx)$ can be generated where ux is not a valid user, by specifying rules for ux . Such tests should of course fail for any value of idx , and rules for choice of idx can be specified. An alternative is to specify sets *Uinvalid* and *Pinvalid* and generate all tests for

$$(ux, idx) : Uinvalid \times Pinvalid.$$

7 Intermediate States

So far only functional properties, like those exemplified by Webbo, have been considered. But it is important for a specifier, and tester, to take account of so-called ‘non-functional’ properties like security, non-interference, distribution and even efficiency. This section considers one common situation that is typically a source of insecurity in the sense of exposing information that ought to remain concealed.

7.1 Sequential Components

The linking of the code to the model implicitly involves different levels of atomicity. It may be convenient, for example, to implement an operation op with a sequential composition $op = op0 \circ op1$ where $op0$ is responsible for some of the I/O and $op1$ for the others. However such a factorisation may result in a security leak.

For example, the login operation of Webbo assumes that the system responds, atomically, only after both the user name and password have been input and the login button has been clicked. But in some implementations, login could be achieved as the sequential composition of two ‘sub’-operations the first of which provides the user with information about the success of part of the process. If that information ought to be secure, then this factorisation of login should be flagged as a security hazard.

More explicitly, login may be expressed as a check that the user identified by input $u?$ is valid, *i.e.* $u? \in U$, (with no output but with a Boolean variable to record $(u? \in U)$ for use by the second sub-operation) followed by a check that if so then the password provided is correct, *i.e.* $log(u?).pd = p?$, (with appropriate change of state and output *ack!*) then an opportunity is provided for a malicious user to obtain information about U by probing with the first sub-operation.

Assume that `loginFail(u?,p?)` is linked to

```
enter_text(:name,u?);
enter_text(:password,p?);
login.click();
check-not-logged-in();
```

with the assumption only that the user $u?$ and the password $p?$ do not tally. It is desired also to check that if $u?$ is not a valid user, no message is returned. To simplify the specification of the link, `loginFail(u?,p?)` is split into a sequential composition

```
loginFailName(u?) ; loginFailPassword(p?) .
```

The code that links the action word `loginFailName(u?)` to the system is

```
enter_text(:name,u?);
check-no-message();
```

and the code that links the action word `loginFailPassword(p?)` to the system is

```
enter_text(:password,u?);
login.click();
check-not-logged-in();
```

An alternative solution to this problem is to have another action word `loginFailNoUser`. But that complicates the specification of the model and the link between the model and code.

7.2 Principle

In general if the code associated with an action word a is of the form $b_0;b_1$,

$$a \rightsquigarrow b_0;b_1,$$

where b_0 represents a desired behaviour in itself, it is better to split a into $a_0;a_1$ and associate a_0 with b_0 and a_1 with b_1

$$a = a_0;a_1, \quad \text{where } a_0 \rightsquigarrow b_0 \text{ and } a_1 \rightsquigarrow b_1.$$

Similarly, if the code associated with an action word a is of the form $b_0;b_1$ where b_1 is associated with another action word b , it is better to associate a with only b_0 and change the model by replacing the a transitions with $a;b$ transitions.

8 Conclusion and Further Work

A model, in the form of a (state-based) specification, together with a link is able to generate tests of a putative implementation. It has been seen that, roughly, the more detailed a test is, the more information the tester needs to have of the link. Thus laws concerning the operations may involve little information beyond the specification; but they might require knowledge of how the link represents the system state in the implementation. The trade-off between specification and implementation knowledge has already been revealed by Stocks [22, 23], so it is not surprising that it has reappeared here.

A specifier might ask the question: what extra functional information am I in a position to provide about the system that yields the kind of redundancy necessary for testing? Various answers have been discussed here, including: system invariants (explicit or implicit), further confidence conditions, partiality of operators, boundary testing and ‘nonfunctional’ properties like security.

The specifier is well placed to augment the specification with various pieces of information—along those lines—to generate tests. But the more detailed the test, the more information is required beyond mere specification information. For otherwise the result might be a test which is not able to be executed due to restrictions on the testing interface. That is particularly true of web-based systems of the kind represented by Webbo.

This paper represents work very much in progress. It is hoped to continue it by formalising the action-word approach to linking specification with implementation (using the concepts of data simulation and process algebra), automating the ideas contained here, applying them to Webbo to determine the nature of the tests generated, and identifying conditions sufficient for a test interface to execute a given family of tests.

References

- [1] B. K. Aichernig. Test-design through abstraction — A systematic approach based on the refinement calculus. *Journal of Universal Computer Science*, 7(8):710–735, 2001.
- [2] R. V. Binder. Design for testability in object-oriented systems. *Commun. ACM*, 37(9):87–101, ACM Press, 1994.
- [3] C. Boyapati, S. Khurshid and D. Marinov. Korat: automated testing based on Java predicates. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, Rome, Italy, IEEE, 22–24, 2002.
- [4] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*. LNCS 3472, Springer Verlag, 2005.
- [5] H. Buwalda. Action Figures. *Software Testing and Quality Engineering*, March/April:42–47, 2003.
- [6] H. Buwalda and M. Kasdorp. Getting automated testing under control. *Software Testing and Quality Engineering*, November/December:39–44, 1999.
- [7] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In J. C. P. Woodcock and P. G. Larsen, editors: *FME'93: Industrial-Strength Formal Methods*, LNCS 670:268–284, Springer Verlag, 1993.
- [8] R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z*. Macmillan Press, 2000.
- [9] I. K. El-Far and J. A. Whittaker. Model-based software testing. In J. J. Marciniak, editor: *Encyclopedia of Software Engineering*, volume 1:825–837, Wiley-Interscience, 2002.
- [10] <http://portal.etsi.org/mbs/Testing/conformance/conformance.asp>
- [11] I. J. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, SE-12(1):124–133, 1986. See also the 2006 revision <http://www.itue.uq.edu.au/~ianh/Papers/spectest.pdf>.

- [12] S. Helke, T. Neustupny and T. Santen. Automating test case generation from Z specifications with Isabelle. In J. P. Bowen, M. G. Hinchey and D. Till, editors: *ZUM'97: The Z Formal Specification Notation*, LNCS **1212**:52–71, Springer-Verlag, 1997.
- [13] H.-M. Hörcher and J. Peleska. The role of formal specifications in software testing. *Tutorial Notes for the FME'94 Symposium*. Formal Methods Europe, 1994.
- [14] H.-M. Hörcher. Improving software tests using Z specifications. In J. P. Bowen and M. G. Hinchey, editors: *ZUM'95: The Z Formal Specification Notation*, LNCS **967**:152–166, Springer Verlag, 1995.
- [15] H.-M. Hörcher and J. Peleska. Using formal specifications to support software testing. *Software Quality Journal*, **4**:309–327, 1995.
- [16] J. Kohl and P. Rogers. Watir Works. *Better Software*, 40–45, April, 2005.
- [17] P. Krishnan. Uniform descriptions for model based testing. *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*, IEEE, 2004. Available at http://epublications.bond.edu.au/infotech_pubs/8.
- [18] D. Li and B. K. Aichernig. Automatic Test Case Generation for RAISE. Technical Report 273, UNU-IIST, P.O.Box 3058, Macao, January 2003.
- [19] E. Mikk. Compilation of Z specifications into C for automatic test result evaluation. In J. P. Bowen and M. G. Hinchey, editors: *ZUM'95: The Z Formal Specification Notation*, LNCS **967**:167–180, Springer Verlag, 1995.
- [20] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.
- [21] The RAISE Method Group. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall, 1995. Available by ftp from ftp://ftp.iist.unu.edu/pub/RAISE/method_book.
- [22] P. A. Stocks. *Applying Formal Methods to Software Testing*. PhD thesis, Department of Computer Science, The University of Queensland, St. Lucia 4072, Australia, 1993.
- [23] P. A. Stocks and D. A. Carrington. A Framework for Specification-based Testing. *IEEE Transactions in Software Engineering*, **22**(11):777–793, 1996.
- [24] B. Legeard, F. Peureux and M. Utting. Automated boundary testing from Z and B. In L. H. Eriksson, P. Lindsay, editors: *Formal Methods Europe, FME 2002*. LNCS **3291**:21–40, Springer-Verlag, 2002.
- [25] M. Utting. Position paper: model-based testing. *Verified Software: Theories, Tools, Experiments*. ETH Zürich, IFIP WG 2.3, 2005.
- [26] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.