January 2004

Uniform Descriptions for Model Based Testing

Padmanabhan Krishnan
Bond University, Padmanabhan_Krishnan@bond.edu.au

Uniform Descriptions for Model Based Testing

Padmanabhan Krishnan
School of Information Technology
Bond University, Gold Coast QLD 4229
Australia
Email: pkrishna@staff.bond.edu.au

Abstract

In this paper a framework which uses linear time temporal logic and model checking techniques to describe the semantics a variety of test specifications is developed. These include a semantics for action words, which are a practical approach to model based testing, and coverage requirements. Features of tools that support the development of tests using this approach are also presented. While model-checking ideas are used, a model of the system is not actually required. Test sequences are directly generated from the specification of properties.

1 Background

It is recognised that manual testing of software is tedious and expensive. Automating the testing process is therefore desirable. However automation of testing is a difficult task. Reasons for this include the sensitivity of the process to changes in the system and what is being tested is not always clear from the test scripts [5, 6]. Deriving the appropriate test sequences is one of the obstacles in automated testing. Model based or specification based testing is a promising technique to generate test cases. Typical examples of this approach include [4, 6, 8, 12]. In this approach a specification or a model of the system is created. Test cases are derived from this specification or model. As a model is usually much smaller than the system it provides a manageable description of the testing process.

Formal methods can be combined with model based specification and testing. Bowen et al. [3] outline a number of possible approaches to formal methods and testing. More specific examples of describing models for testing include using Z [13], abstract state machines [12], automata represented as graphs called test graphs [17], or as tables representing behaviour [11]. Model checking [7] has also been used for testing [14]. Model checking determines if a formal model (e.g., a finite state automaton) satisfies a given specification usually specified in a temporal logic. If the model does not satisfy the requirement counter examples in terms of the model (e.g., a run of the automaton) are generated. By negating the appropriate requirements various counter examples can be generated. These counter examples can then form the basis of test sequences.

The process of model checking requires both a model and a specification. This is unlikely given the overheads in creating specifications and models. Garganti and Heitmeyer [11] use model checking to generate test sequences from an operational model expressed in SCR. But they do not require a specification. The operational model can be viewed as specifying the tests for the actual implementation.

To ease the automation of the testing process a separation of the internal representation and the high level aim of the test has been suggested. This is especially useful for testing GUIs [6]. Jia and Liu [15] use first order logic formulae and their representation in XML for testing Web applications. They also use XPath expressions as basic predicates. As one of the aims is to test returned HTML pages, they also provide matching operators such as `startsWith`, `contains` and `endsWith`. Using first order quantification over predicates and operators a wide variety of desired properties can be specified. A hierarchy of test suites, which consist of test cases which are further

Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)
1530-0803/04 $ 20.00 © 2004 IEEE

decomposed into test steps is used. The XML representation is then linked to other testing tools such as JUnit to actually implement the test. Ricca and Tonella [18] adopt a similar strategy for white-box testing of web pages. They use path expressions over a model of web pages to automatically generate test sequences.

The action word approach [5, 4] has also been proposed to simplify test maintainability. This is achieved by separating the test specification from the actual testing. That is, the design of the test is specified at a high level with various scripts and translators generating and running the appropriate tests from the design. An action word specifies a sequence of actions such as entering a value or generating an event or checking an outcome. The framework permits grouping of action words to define a test cluster. Test clusters are logically related tests and specify "what is being tested" rather than "how are things tested". The emphasis is on developing scenarios which are used for testing. This can be related to the SCR notation [11] where event tables are used. The practical effectiveness of scenario based testing over general model based testing has been argued by Buwalda [4].

The use of satisfaction in propositional logic to generate input output traces as test sequences has been reported by Wimmel et al. [20]. The model is constructed as a finite state automaton but the tests themselves are finite. This approach can be viewed as an application of bounded model checking. The authors claim that the advantage of the bounded model checking approach is that temporal logic formulae (e.g., linear time temporal logic (LTL) or computation tree logic (CTL)) are not required as they are harder to understand. The use of specification patterns [9] simplifies the use of temporal logic. The patterns, which are easier to use, provide templates for frequently required properties. The translation of these patterns into the underlying basic logic is straightforward. Coverage is an important criteria for testing methodologies. A state machine based approach to coverage is developed by Friedman et al. [10]. The required coverage criteria are expressed as logical properties and an extension of model checking is used to generate the test sequences.

In summary, most model based testing techniques use states and transitions in a finite automaton to generate test sequences. In some approaches, e.g., [6], these automata are explicitly constructed while in others, e.g., [11], they are described in a logical fashion. In this paper the aim is to develop a general framework for expressing a wide variety of testing models. For instance, the state coverage requirements and the action word approach can be combined with the specification patterns to generate appropriate testing requirements. Furthermore by translating these diverse test specifications to a suitable logic (linear time temporal logic or LTL), automata theoretic and model checking ideas for testing can be exploited. If a model is provided model-checking will yield counter examples which can be used for testing. Otherwise, the LTL formula can be converted directly to a Büchi automaton which can be used to generate sequences. Thus the main contribution of this work is to enable a practitioner to use formal specification based testing. This is because the tester need not learn the formal syntax and semantics of the temporal logic and the automation of test generation using automata theoretic techniques. Another contribution is the option of using a model along with the specification to generate test sequences. Tools which assist in this are also described. The overall picture is given in Figure 1. It indicates that in principle a variety of inputs can be accepted from testers and the input can be translated to LTL. If a formal finite state model is available, traditional model checking techniques can be used. Otherwise, the LTL formula can be converted to an Büchi automaton from which test sequences can be generated.

The reason for using temporal logic as the formal language for testing is that it is usually a compact way of representing automata. LTL is used as it permits the specification of sequences that can be related to testing. It can also be related to paths such as those used in XPath. An abstract description of various XPath descriptions has been developed by Benedikt et al. [1]. While the paper emphasises axiomatisation of various fragments, the basic abstract definition can be used in conjunction with action words, and temporal logic patterns to provide an expressive language to express testing requirements. This can then subsume the work reported by Jia and Liu [15].

In the next section a brief overview of LTL, action words, issues in web testing and state coverage specifications is given. The focus is on action words and

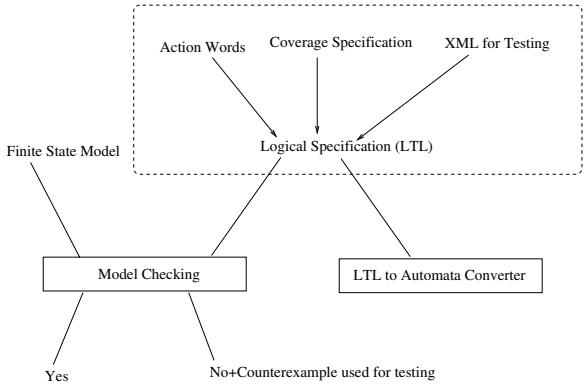

Figure 1: General Structure.

web testing as that has the most practical significance. Section 3 the overall syntax to develop models for testing and the main translation scheme is described. Examples using the translation scheme are developed in Section 4.

2 Preliminaries

As with any temporal logic, the basic units are predicates which describe properties at a given state. These predicates are built up from variables, values and operators from the application domain. For example, in the action word approach comparing variables with other variables or values are the atomic predicates. For web testing the values are URLs and codes to indicate responses to http requests while operators include the XPath matching operators of `contains`, `startsWith`, `endsWith` etc. User interface testing uses operations such as clicking on values such as buttons or entering values into forms. As the aim is to be as general as possible, the exact syntax for defining the predicates is not prescribed. The existence of a syntactic category (say BP) for basic predicates is assumed. The state predicates (represented by Predicate) are then boolean combinations (using the usual $\wedge$, $\vee$ and $\neg$) over terms from BP. Standard abbreviations for implies $\rightarrow$, equivalence $\equiv$ etc. are also assumed.

Linear time temporal logic (LTL) is a temporal logic [19] which describes properties over infinite sequences. The behavioural properties are boolean combinations of two types of temporal forumulae . The modality $\bigcirc$ (or 'X') represents the next state in sequence while the modality $\mathcal{U}$ (or 'U') represents until. The syntax used is given below.

Beh ::= Predicate | Beh '$\wedge$' Beh |
Beh '$\vee$' Beh | '$\neg$' Beh |
'X' Beh | Beh 'U' Beh

The standard abbreviations of 'G' for globally true (i.e., over the entire trace), 'F' for finally or eventually true can also be used. 'F φ' is defined as 'true U φ' while 'G φ' is defined as '$\neg$ F $\neg$ φ'. Other abbreviations such as those defined in [9] will be introduced when required.

2.1 Action Words

A test cluster is a collection of tests that have similar goals and level of detail. The clustering represents the logical structuring of the tests. In the action word approach [5] a test cluster is represented as a spreadsheet. Such a spreadsheet consists of a collection of independent tests as well as sequential tests. All tests in the independent test collection are run from the same starting point. These enable the testing of possible choices at a given state. The sequential tests are run one after the other in order and test a particular behaviour of the system.

This basic model has been extended to represent decision tables where there is an explicit notion of input and output. From these tables particular testing scenarios are generated. The advantages of this approach are that the testing models are easy to define and it is possible to generate the test sequences. However these tables do not have a well defined semantics. For example, there is no clear distinction between independent and sequential tests. Also the separation of inputs and responses is not always defined in the table.

The example in [4] of testing an insurance system is discussed below. Table 1 describes a decision table linking the number of accidents to gender. This table can be referenced in other tables which be used in action words for testing. For instance, the last entry shown in the Table 1 indicates a new test table which refers to values in other tables. The test sequences generated correspond to inserting various values of accidents and gender from the decision table. The # indicates a table reference. That is, the values are obtained

accidents	gender	situation
0	–	accept
2	f	accept
2	m	refuse
3	–	refuse

enter customer	Chris	#accidents	#gender

Table 1: Decision Table.

from the corresponding table. In keeping with the row semantics the value used for accidents and gender correspond to elements in the same row. However, it is not evident from the table whether these tests are independent or sequential. The constructs presented in this paper enable such distinctions and hence permit precise definitions for testing.

The tabular approach can be used to specify automata. By using keywords such as `move`, `from` and `to` transitions in a finite state automaton can be defined. The resulting automaton represents the model to generate test sequences. As such automata are manually defined they are necessarily small and notions such as coverage are not addressed in the action word approach.

2.2 GUI and Web Testing

The testing of graphical user interfaces is difficult because issues such as dynamically created interfaces and interpreting user actions (e.g., clicking buttons, filling in forms) need to be handled. These are usually tied to the internal representation which makes automation difficult. The capture/replay technique is usually adopted to develop test scripts which run tests derived from the captured usage. However as shown in [6] it is possible to build an abstract representation of the user interactions as a finite automaton which can then be used for testing. Testing web-based applications is similar as pages can be dynamically generated. Model-based testing of web pages can involve a sequence of web pages to visit which are either linked via hyperlinks or generated dynamically. A test suite can be specified as a path following various links within the set of pages associated with the web site [18]. For instance, a path can be defined as $l_1 l_2 (l_3 + l_4)$ which requires the test to follow links l_1 and l_2 and then either follow l_3 or l_4. In order to follow these links suitable values for input forms may need to be generated. A more concrete approach has been taken in [15]. They explicitly specify requesting a URL and checking the results with the match operators. For example, `request url=http://www.cs.depaul.edu/program` requires the specified URL to be requested. A requirement of the form

```
select="/html/body"
match op="contains"
value="Undergraduate"
```

and `match op="contains" value = "Master"` for the response demands the body of the web page identified by the URL to contain the values `Undergraduate` and `Master`. They also allow quantification over values. For example, the specification

```
forall variable name="x"
match op="startswith"
select="$x" value="http"
```

requires all URLs in the given page to be absolute and begin with `http`. Here the quantification is over a new variable called `x`.

2.3 Coverage Specifications

Coverage specifications at the code level deal with statements and branching. It has been argued by Benjamin et al. [2] that path coverage yields better results than transition or state coverage. While creating coverage models for testing there are three categories of specifications. The first category specifies the variables in the system that can be ignored to create the model. The second category specifies the variables for which the test cases should cover all possible values. The third category specifies the variables along with a set of values to be covered by the test cases. However, a model is an abstraction of the code and coverage requirements as specified in [10] deal with both the abstraction as well as requirements on the model. For instance, the specification `State_Projection` b ON e generates test sequences for each state in the model identified by the possible values for the expression e where b is true. Specifications for transitions

which identify pairs of states are also possible. Both these type of requirements specify which states or transitions must be visited. The dual of these specifications which forbid visiting certain states or transitions is also possible. For instance `Forbidden_State` b results in the exclusion of states where b is true as well as all other states that are reachable only from these excluded states. The forbidding of states allows one to control the states in the model as well as generating tests which skip parts under development.

3 Syntax, Tool Support and Semantics

The abstract syntax for describing a variety of tests is given in Figure 2. Informally a predicate (say s) requires the current state to satisfy s. The connective ';' specifies sequencing with the second test run from the state where the first test terminates. The connective '+' specifies the running of the two tests one after the other but from the same starting state. That is, they are different test runs. The STATE, NOSTATE, PATH and NOPATH connectives are used to specify the desired coverage requirements of the tests. The 'Step' construct specifies if the tests are run from the next state or some state reachable after a sequence of actions. These are directly influenced by the LTL modalities. The 'Step' specifications are required as the real system may perform a number of steps before exhibiting the actions specified in the model. In this paper we focus only on these constructs. However, they can be extended with the patterns [9] as the patterns have a suitable LTL semantics.

Test ::= Predicate |
Test ; Step Test | Test + Test |
STATE Step Test |
NOSTATE Step Test |
Test Step PATH Test |
Test NOPATH Step Test
Step ::= IMMEDIATE | FUTURE | EVERY

Figure 2: Abstract Syntax.

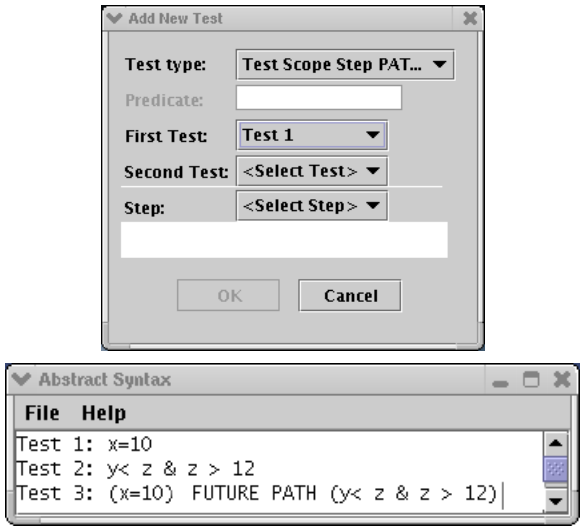

Figure 3: Abstract Syntax Tool.

3.1 Tool Support

A proof of concept tool, written in Java, to ease the development of these tests has been designed. The tool can be used in two ways. The first is to directly manipulate the tests described using the abstract syntax. A pull down list can be used to create tests belonging to the various categories. The created lists are added to another list which can then be edited or used for other tests. The list of tests is saved as a text file which enables writing of scripts to interface it with other tools. An example is shown in Figure 3.

The tool also has a GUI where icons for the various syntactic elements are used. The main aspects include using different arrow types for the three types of steps and colour to distinguish negation from positive predicates. Predicates are also defined using a tabular fashion (formula are in the disjunctive normal form) keeping it close to the action word approach. Examples are shown in Figure 4. The first item shows a predicate in tabular form where the darker colour (red on the screen) indicates negation. The second item shows the sequencing of tests (hence the second test is run even if the formula corresponding to the first test does not hold and is shown as half green and half red) The step is Future indicated by an arrow followed by a dot. The third item shows a global (indicated by the large arrow head) coverage requirement and corresponds to the textual STATE EVERY construct.

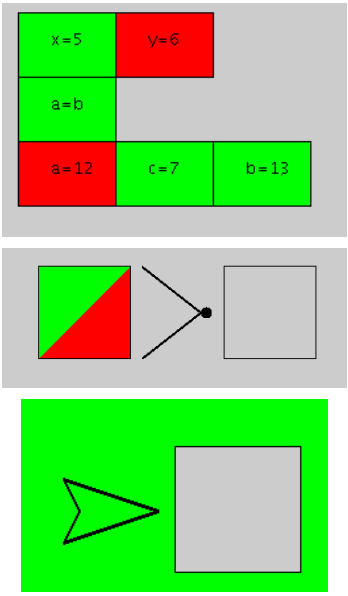

Figure 4: Graphical Tool.

The tool used to generate test sequences will be described after describing a few examples. This will illustrate the applicability of the chosen tool.

3.2 Semantics

The precise semantics in terms of LTL definitions are developed below. The semantics are indexed by formula which is necessary to describe sequential tests. To simplify the presentation $\overline{s}$ is used to indicate the semantics of the Step s. The semantics of a Step yields a LTL modality. That is, $\overline{\text{IMMEDIATE}}$ yields **X**, $\overline{\text{FUTURE}}$ yields **F** and $\overline{\text{EVERY}}$ yields **G**.

$[\![p]\!]^{\varphi} = p \wedge \varphi$

The indexed semantics of a state predicate is the predicate itself and the index formula. This index is usually a formula about the future and hence describes the behaviour of the test after the predicate has become true.

$[\![t_1;\ s\ t_2]\!]^{\varphi} = [\![t_1]\!]^{\varphi_1}$ where $\varphi_1 = \overline{s}\ [\![t_2]\!]^{\varphi}$

The semantics of ';' is sequencing. After the composite test $t_1; s\ t_2$ is run the test corresponding to φ needs to be executed. This means that the semantics of the test to be run after t_1 is the semantics of t_2 with the appropriate scope and test followed by φ.

$[\![t_1 + t_2]\!]^{\varphi} = [\![t_1]\!]^{\varphi} \wedge [\![t_2]\!]^{\varphi}$

As the tests t_1 and t_2 are independent the combined test is represented by conjunction. The test represented by φ will be run after both t_1 and t_2.

$[\![\text{STATE}\ s\ t_1]\!]^{\varphi} = \overline{s}\ [\![t_1]\!]^{\varphi}$

The step identifies the state from where the test t_1 will be executed.

$[\![\text{NOSTATE}\ s\ t_1]\!]^{\varphi} = \overline{s}\ (\neg\ [\![t_1]\!]^{\varphi})$

The NOSTATE requirement specifies the set of states that should not be visited. The step specification defines where in relation to the current state these forbidden states should not occur.

$[\![t_1\ s\ \text{PATH}\ t_2]\!]^{\varphi} = [\![t_1]\!]^{\texttt{True}} \rightarrow \overline{s}\ [\![t_2]\!]^{\varphi}$

The PATH requirement specifies that if t_1 is applicable then there is a path to t_2. As the sequencing of tests applies only after t_2 is executed, the semantics of t_1 is indexed by `True`. Thus if t_1 does not hold in a particular run, the next test run is `True` which holds trivially.

$[\![t_1\ \text{NOPATH}\ s\ t_2]\!]^{\varphi} = [\![t_1]\!]^{\texttt{True}} \rightarrow \overline{s}\ \neg\ [\![t_2]\!]^{\varphi}$

The NOPATH specification is similar to PATH except that t_2 is not applicable. As for NOSTATE the step identifies where the forbidden test applies.

4 Examples

In this section the translation from other test specification languages to a specification following the syntax described above is described.

The first focus is on web-based testing. The key to the translation is to define sufficient predicates which capture behaviour. For example the predicates associated with web pages include `visited` (which indicates if a page has been visited) while the predicates associated with links include `request`, `responseValid` and `responseInvalid`.

The test specification corresponding to the requirement of visiting every page is specified as the conjunction over

STATE FUTURE `visited(p)`

for every page `p`. The specification corresponding to every link is valid is specified as the conjunction over

STATE FUTURE `requested(l)`; FUTURE `responseValid(l)`

for every link `l`. The formal semantics of the above requirement is F (`requested(l)` $\wedge$ F `responseValid(l)`). That is, the link `l` is requested sometime and sometime after that there is a valid response. This is different from

STATE FUTURE `requested(l)` PATH FUTURE `responseValid(l)`

where it does not demand that the link is actually requested. It only requires that if the link `l` is requested the response is valid.

Pages or hyperlinks can also be described by explicit URLs and responses from the server as integer codes (such as 200, 404). Hence variables such as `reqURL` and `respStatus` which can take appropriate value can also be defined. Using these variables, the requirement that every time one requests a particular URL (say `u`) will return a valid status, can be described as

STATE EVERY (`reqURL = u`); FUTURE (`respStatus = 200`)

The modelling of XPath based techniques requires the introduction of suitable functions for each selection and match operation. For example, if the selection is `body` and the operation is `contains` a function `bodyContains` is defined. It is a function from the set of appropriate values to a boolean value. Thus if the test requirement then specifies a value (say `v`) the predicate defined by `bodyContains(v)` can be used in the specification.

The action words described in [5] have a direct translation using state predicates and the ';' connective.

	last name	account	
enter customer	Wood	12345	
	account	last name	
check name	12345	Forest	Incorrect

Table 2: Invalid Value.

accept	check premium = 1000
refuse	check message = Not Eligible

Table 3: Expected Output.

Consider the example in Table 2. It specifies a behaviour which ensures that the values entered are correctly stored in the system. After creating a new customer Wood with account number 12345, the name associated with the account cannot be Forest.

The predicates are `enterCustomer`, `checkName`. The variables are `lastName` and `account`. The specification of the above test can be described as

(`enterCustomer` $\wedge$ `lastName=Wood` $\wedge$ `account=12345`);
EVERY `checkName` $\wedge \neg$ (`lastName = Forest` $\wedge$ `account=12345`)

This translation cannot be automated as there is no uniform syntax to show incorrect behaviour. The second part of the test in the translation presented here states that the account number and the last name as per the input do not tally.

The translation of the more complicated action words [4] follows a similar approach. The only exception is that when a table reference is encountered a sequence of tests using '+' is created. Table 3 combined with the Table 1 captures the example on page 46 of [4]. The values in Table 3 indicate the expected output from the system.

A systematic translation of these tables resulted in the following formulae being generated.

- G (accept $\Rightarrow$ premium=1000) $\wedge$ G (refuse $\Rightarrow$ message="Not Eligible").

That is, if the request is accepted the premium costs 1000 otherwise the message "Not Eligibile" is displayed.

- G ((customer=chris $\wedge$ accidents = 0) $\Rightarrow$ F accept).

  If the person has had no accidents then the request for insurance is granted.

- G ((customer=chris $\wedge$ accidents = 2 $\wedge$ gender=f) $\Rightarrow$ F accept) $\wedge$ G ((customer=chris $\wedge$ accidents = 2 $\wedge$ gender=m) $\Rightarrow$ F refuse).

  If the person has had two accidents then the request for insurance is granted only if they are female.

- G ((customer=chris $\wedge$ accidents = 3) $\Rightarrow$ F refuse).

  If the person has had three accidents then the request for insurance is always denied.

These are all independent tests and they are combined using conjunction.

In the case when testers do not explicitly provide a model, test sequences can be generated using Büchi automata. The LTL formula corresponding to the test can be used by a tool such as LTL2BA `http://www.liafa.jussieu.fr/~oddoux/ltl2ba/` which can generate the automaton corresponding to the formula. LTL2BA provides a number of parameters to generate the automaton and it is possible to generate a compact automaton.

For example, the automaton corresponding to 'G ((customer=chris $\wedge$ accidents = 3) $\Rightarrow$ F refuse)' is shown in Figure 5. In the figure `p` stands for 'customer=chris', `q` represents 'accidents=3' and `r` represents `refuse`. The automaton is non-deterministic as it can move to the state marked '1' at any time. However, for accepting runs, it should move to the state marked '1' only when it cannot remain in the state marked init. The test sequences from the automaton could include making `p` and `q` true and `r` false which will cause the transition to state `1`. The automaton will remain there till `r` is true.

A more complicated example is derived from the 'false in between' pattern described in [9]. This is related to the test in Table 2. The test implicitly assumes that no name change occurs in between entering the value

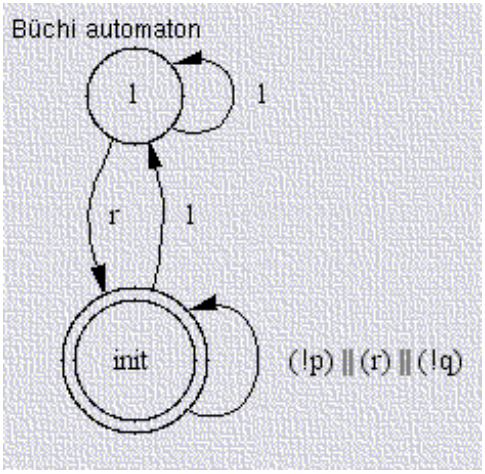

Figure 5: Generated Automaton.

and checking the result. The test sequences for this are generated from the following automaton output by LTL2BA shown in Figure 6. The test is specified by 'No p between q and r' where q represents creating the value, r represents checking the value and p represents updating the name. The LTL formula is given by 'G (q $\Rightarrow$ (($\neg$ p U r) $\vee$ G ($\neg$ r))). The automaton generated has two main cases. In the first case, an 'r' does occur and hence after a 'q' a 'p' should not occur. In the second case, an 'r' never occurs so there is no restriction on 'p'.

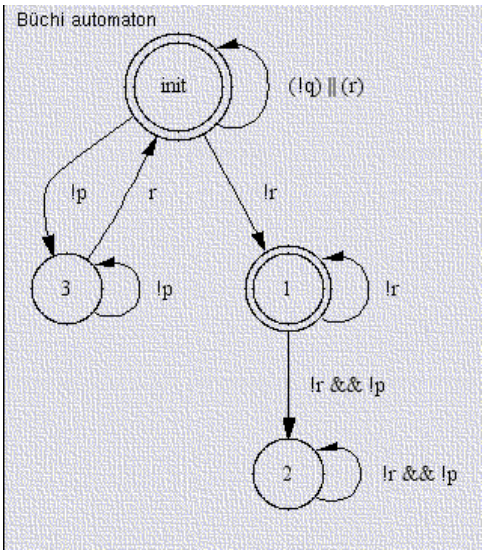

Figure 6: False in Between Automaton.

The test sequences generated fall into two categories.

One which is similar to the old test (with no intervening change to the name). The other is where the a name change occurs and the test succeeds trivially.

5 Conclusions and Future Work

In this paper an uniform approach to developing model based test specifications has been developed. The starting point was the informal but practical action word approach. The formal semantics for the specification language was used to make the semantics of action words based testing more precise. The notion of coverage was also added to the basic action word approach. The temporal logic approach removes the need to explicitly specify test automata (e.g., as in [6]). The test specifications are more compact. The use of LTL based modalities and predicates has enhanced the reuse of many test specifications. In a system under development many tests represented as event sequences (e.g., [16]) are made unusable after changes are made. However the use of connectives such as FUTURE ensure that changes are not required as not all intermediate steps are represented. For example, the same test specifications are applicable to testing two spreadsheets (the Open Office Calc and gnumeric) although the key menus are different.

The tools described in this paper provide only proof of concept and need to be further developed. A well developed tool should also be integrated with the LTL2BA tool and model checkers. One of the current goals is to extend the framework to include timing constraints. This will enable the specification of stress tests and other performance related tests [21].

Acknowledgments

The author thanks Shane Bracher for his programming support and Phil Stocks for his valuable comments on an earlier version of the paper.

References

[1] M. Benedikt, W. Fan, and G. Kuper. Structural Properties of XPath Fragments. In *International Conference on Database Theory*, volume 2572 of *LNCS*, pages 79–95. Springer, 2003.

[2] M. Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, and Y. Wolfsthal. A study in coverage-driven test generation. In *Proceedings of the 36th ACM/IEEE conference on Design Automation Conference*, pages 970–975, 1999.

[3] J. P. Bowen, K. Bogdanov, J. Clark, M. Harman, R. Hierons, and P. Krause. FORTEST: Formal methods and testing. In *Proc. COMPSAC 02: 26th IEEE Annual International Computer Software and Applications Conference*, pages 91–101, Oxford, UK, August 2002. IEEE Computer Society Press.

[4] H. Buwalda. Action figures. *Software Testing and Quality Engineering*, pages 42–47, March/April 2003.

[5] H. Buwalda and M. Kasdorp. Getting automated testing under control. *Software Testing and Quality Engineering*, pages 39–44, November/December 1999.

[6] J. Chen and S. Subramaniam. Specification-based Testing for GUI-based Applications. *Software Quality Journal*, 10:205–224, 2002.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[8] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In *Formal Methods Europe*, LNCS 670, pages 268–284, 1993.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *21st International Conference on Software Engineering*, pages 411–420, 1999.

[10] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *Proceedings of the international symposium on Software testing and analysis*, pages 134–143. ACM Press, 2002.

[11] A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering–ESEC/FSE*, LNCS 1687, pages 146–162, Toulouse, France, 1999. Springer Verlag.

[12] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proceedings of the International symposium on Software testing and analysis*. ACM Press, 2002.

[13] R. Hierons. Testing from a Z specification. *Software Testing Verification and Reliability*, 7(1):19–33, 1997.

[14] T. Jéron and P. Morel. Test generation derived from model-checking. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–121, Trento, Italy, July 1999. Springer-Verlag.

[15] X. Jia and H. Liu. Rigorous and automatic testing of web applications. In *IASTED Conference on Software Engineering and Applications*, pages 280–285, November 2002.

[16] A. Memon and M. Soffa. Regression Testing of GUIs. In *ESEC/FSE*, pages 118–127, 2003.

[17] T. Miller and P. Strooper. Supporting the Software Testing Process through Specification Animation. In *Software Engineering and Formal Methods*, pages 14–23. IEEE, 2003.

[18] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE-01)*, pages 25–36. IEEE, May12–19 2001.

[19] C. Stirling. Modal and temporal logics. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 477–563. Oxford Science Publications, 1992.

[20] G. .Wimmel, H. Loetzbeyer, A. Pretschner, and O. Slotosch. Specification based test sequence generation with propositional logic. *Special Issue on Specification Based Testing, Journal on Software Testing, Validation, and Reliability (STVR)*, 10(4):229–248, 2000.

[21] J. Zhang and S. C. Cheung. Automated Test Case Generation for the Stress Testing of Multimedia Systems. *Software Practice and Experience*, 32:1411–1435, 2002.