Rob Hierons and Thierry Jéron (Eds.)

Formal Approaches to Testing of Software

FATES'02
A Satellite Workshop of CONCUR'02
Brno, Czech Republic, August 24th 2002
Proceedings

Preface

Testing is an important technique for validating and checking the correctness of software. However, the production and application of effective and efficient tests is typically extremely difficult, expensive, laborious, error-prone and time consuming. Formal methods are a way of specifying and verifying software systems by applying techniques from mathematics and logic. This enables the developer to analyze system models and reason about them with mathematical precision and rigour. Thus both formal methods and software testing can be used to improve software quality.

Traditionally, formal methods and testing have been seen as rival approaches. However, in recent years a new consensus has developed. Under this consensus, formal methods and testing are seen as complementary. In particular, it has been shown that the presence of a formal specification can assist the test process in a number of ways. The specification may act as an oracle or as the basis for systematic, and possibly automatic, test synthesis. In conjunction with test hypotheses or a fault model, formal specifications have also been used to allow stronger statements, about test effectiveness, to be made. It is possible that the contribution to testing will be one of the main benefits of using formal methods.

The aim of the workshop FATES — *Formal Approaches to Testing of Software* — is to be a forum for researchers, developers and testers to present ideas about and discuss the use of formal methods in software testing. Topics of interest are formal test theory, test tools and applications of testing based on formal methods, including algorithmic generation of tests from formal specifications, test result analysis, test selection and coverage computation based on formal models, and all of this based on different formal methods, and applied in different application areas.

This volume contains the papers presented at FATES'02 which was held in Brno (Czech Republic) on August 24, 2002, as an affiliated workshop of CONCUR'02. Out of 17 submitted papers the programme committee selected 9 regular papers and 1 position paper for presentation at the workshop. Together with the keynote presentation by Elaine Weyuker, from AT& T Labs, USA, they form the contents of these proceedings.

The papers present different approaches to using formal methods in software testing. The main theme is the generation of an efficient and effective set of test cases from a formal description. Different models and formalisms are used as the starting point, such as (probabilistic) finite state machines, X-machines, transition systems,

categories, B, Z, Statecharts, UML, and different methodologies and algorithms are discussed for the test derivation process, ranging from formalization of the manual testing process to the (re)use of techniques from model checking.
The papers give insight in what has been achieved in the area of software testing with formal methods. Besides, they give clear indications of what has to be done before we can expect widespread use of formal techniques in software testing. The prospects for using formal methods to improve the quality and reduce the cost of software testing are good, but still more effort is needed, both in developing new theories and in making the existing methods and theories applicable, e.g., by providing tool support.

We would like to thank the programme committee and the additional reviewers for their support in selecting and composing the workshop programme, and we thank the authors for their contributions without which, of course, these proceedings would not exist.
Last, but not least, we thank Antonin Kucera for arranging all local matters of organizing the workshop, and the Masaryk University of Brno for giving the opportunity to organize FATES'02 as a satellite of CONCUR'02, INRIA for supporting the printing and distribution of these proceedings, and the EPSRC Formal Methods and Testing network.

Brunel and Rennes, August 2002

Rob Hierons
Thierry Jéron

Programme committee

Ed Brinksma	(University of Twente, The Netherlands)
Rocco De Nicola	(Università degli Studi di Firenze, Italy)
Marie-Claude Gaudel	(Université de Paris-Sud, France)
Jens Grabowski	(Universität Lübeck, Germany)
Dick Hamlet	(Portland State University, United Kingdom)
Robert Hierons	(Brunel University, United Kingdom), co-chair
Thierry Jéron	(INRIA Rennes, France), co-chair
David Lee	(Bell Labs, Beijing, China)
Brian Nielsen	(Aalborg University, Denmark)
Jeff Offutt	(George Mason University, USA)
Doron Peled	(University of Texas at Austin, USA)
Alexandre Petrenko	(CRIM, Canada)
Jan Tretmans	(University of Nijmegen, The Netherlands)
Antti Valmari	(Tampere University of Technology, Finland)
Carsten Weise	(Ericsson Eurolab Deutschland GmbH, Germany)
Martin Woodward	(Liverpool University, United Kingdom)

Referees

Boroday Serge,	(CRIM, Canada)
Ebner Michael,	(Universität zu Lübeck, Germany)
Jard Claude,	(IRISA/CNRS Rennes, France)
Le Traon Yves,	(IRISA/University Rennes I, France)
Marchand Hervé,	(IRISA/INRIA Rennes, France)

Local organization

Antonin Kucera,
Masaryk University,
Brno,
Czech Republic

Contents

Thinking Formally About Testing Without a Formal Specification

Elaine J. Weyuker
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
weyuker@research.att.com

Abstract

Practitioners test software every day and have to make decisions about what techniques to use to select test data. This paper discusses what it means for one test data selection criterion to be more effective than another. Several proposed comparison relations are presented and discussed. Deficiencies are highlighted, and a discussion of how these relations evolved is presented. The usefulness of empirical studies is also considered.

1 Introduction

I recently addressed a meeting of roughly 400 software testers. All of them were bright, experienced, knowledgeable, educated. Most were from the telecommunications world - land of the state machine. I asked how many of them routinely saw a formal specification. I included such things as various types of state machines, Statecharts, UML, LOTOS, Estelle, SDL, and Z specifications, or anything else they themselves classified as a formal specification. Instead of seeing a sea of hands in the air, I was instead greeted by a chorus of laughter. Surely I had told a great joke. I then thought that perhaps I was asking too much when I used the term "routinely" and so I modified the question to: "How many have *ever* seen a formal specification?" One lone person raised their hand - he used to work in the switching area and saw it in that arena. This was indeed an eye-opening experience.

It is very common in the telecommunications world to have very high reliability requirements (people often speak of "five nines" or .99999 reliability, although exactly what that means is frequently unclear) and very high availability requirements (so-called 24-7 or always available). We spend a great deal of time and money and effort testing the software. We have many testing stages: unit, feature, integration, system, load, stress, performance, end-to-end, operations readiness to name just some of the levels, and there is a research literature that tells us how good it would be and what to do with a formal specification, and yet the real world that spends the time and money doing the testing doesn't look at that literature by and large, and often isn't even aware of it. So there is a disconnect and it is not clear when or whether this gap will be breached.

Therefore, this paper is *not* about what people traditionally think about when they hear the term "formal methods" associated with software testing: namely testing using a formal specification. Instead it will be about other things that *might* be formally assessed without a formal specification, and that desperately need to be assessed relative to software test case selection strategies. In particular, I will focus on how to compare the effectiveness of testing strategies to help us determine which is the best approach for a given program, or for all programs in general. I will describe various false attempts

at defining comparison relations, and describe where we are now. I will discuss why the proposed relations were not satisfactory. I hope this will challenge some of you to think about this important issue in new ways, and lead to a renewed interest in the problem, and some exciting new insights.

2 Comparison Relations

Probably the first suggestion for how to compare software testing strategies used the subsumption relation. Intuitively, subsumption is a very natural way to compare strategies. It really seems like it captures the essence of what we mean when we say that one testing strategy is more effective or more comprehensive than another.

Formally, criterion C_1 *subsumes* criterion C_2 if for every program P, every test suite that satisfies C_1 also satisfies C_2.

In [11] and [12], Rapps and Weyuker introduced a new family of dataflow-based test case selection strategies, and several controlflow-based and dataflow-based criteria were compared using the subsumption relation. We firmly believed that this relation indicated that the subsumed relations were less good than the subsuming relations.

However, it was later recognized that subsumption had definite deficiencies when used to compare testing strategies. The first deficiency was recognized and pointed out by Rapps and Weyuker in [11], namely that many testing criteria are not comparable using subsumption, in the sense that neither subsumes the other.

An even more serious limitation associated with subsumption is that it can be misleading. It is easy to come up with examples of testing strategies such that C_1 subsumes C_2 but there are test suites that satisfy C_2, the supposed weaker strategy, that expose faults, while test suites that satisfy C_1, the more demanding strategy, do not expose any faults. The problem arises because there are typically many different test suites that satisfy a given strategy and generally there is little or no guidance as to which one to choose. Therefore, you may be lucky when selecting the test suite for C_2 and unlucky when selecting the C_1 test suite, or it may even be that the "natural" test suite to select for C_2 does a better job of uncovering faults than does the "natural" test suite that one would select to satisfy C_1.

With this problem in mind, Gourlay [7] introduced the power relation. A criterion is said to *detect* a failure if *every* test set that satisfies that criterion contains an input that causes P to fail, and there is at least one test set satisfying the criterion for P. Then, criterion C_1 is at least as *powerful* as criterion C_2 if for every program P, if C_2 detects a failure in P, then so does C_1.

Although the intent of this relation was to address subsumption's deficiencies, the power relation still has weaknesses. In particular, the incomparability problem remains since many criteria are still incomparable under the power relation. In addition, the power relation does not entirely eliminate the situation in which the "weaker" relation exposes faults while the "stronger" one does not. One problem is that the power relation is based on the definition of "detecting" a failure which is a very demanding one. If C_1 is at least as powerful as C_2, there may still be failures that will more often be exposed by C_2 than C_1 even though neither criterion detects them in the formal sense.

It may also happen that P contains faults but neither C_1 nor C_2 detects any failures because of the stringent requirements of the notion of detection. As discussed relative to the subsumption relation, most test selection criteria do not require the selection of

specific test cases, so a criterion will be satisfied by many different test sets, some of which will include inputs that fail, while others will not include any inputs that fail. Since this is the case, it is possible that the test set selected for the weaker criterion will expose more faults than the one selected for the stronger one.

Perhaps the most important deficiency is that there is no requirement that the faults uncovered by C_1 and C_2 be the same. Therefore, C_1 might detect nothing but trivial faults while C_2 detects catastrophic faults.

Once again, an attempt was made to address the weaknesses associated with both the subsumption and power relations. In Reference [15], Weyuker, Weiss, and Hamlet introduced the BETTER relation. They first defined the notion of a test case being *required* by a criterion C to test a program P, if every test set that satisfies C must include that test case. Then, criterion C_1 is $BETTER$ than criterion C_2 if for every program P, if any failure-causing input *required* by C_2, is also *required* by C_1. They showed that:

$$(C_1 \; subsumes \; C_2) \Rightarrow (C_1 \; BETTER \; C_2) \Rightarrow (C_1 \; at \; least \; as \; powerful \; as \; C_2)$$

They also proved that the converse did not hold. It therefore follows that these three relations are all distinct relations.

Again the newly-defined relation that was designed to solve the previously-defined relations' problems, had its own problems. As before, the incomparability problem had not been solved, and very few criteria actually *require* the selection of specific test cases. This means that the set of failure-causing inputs *required* by a criterion will typically be empty, even though P contains faults. It is also very difficult to show that one criterion is BETTER than another directly. Often the easiest or only way to show this is to show that the subsumption relation holds, and hence by the above theorem, that the BETTER and power relations hold too.

So we see that although several formal ways of comparing software testing strategies have been proposed and used to do the comparison, reflection indicates that they don't really tell us what we'd like them to tell us about the relative effectiveness of different strategies.

There is another way of noticing that the above-cited types of comparison relations fall short of ideal, and this helped us focus our attention in a different direction. Consider the following simple example. Let P be a program with domain $D = \{0,1,2,3,4\}$. Assume there is only one failure-causing input in the domain, namely 0.

Assume that C_1 requires the selection of one test case from the subdomain $\{0,1,2\}$ and one test case from the subdomain $\{3,4\}$, while C_2 requires the selection of one test case from the subdomain $\{0,1,2\}$ and one test case from the subdomain $\{0,3,4\}$. Then six test sets satisfy criterion C_1: $\{0,3\}$, $\{0,4\}$, $\{1,3\}$, $\{1,4\}$, $\{2,3\}$, and $\{2,4\}$, of which two, ($\{0,3\}$ and $\{0,4\}$), or one-third expose the fault. Nine test sets satisfy criterion C_2: $\{0,0\}$, $\{0,3\}$, $\{0,4\}$, $\{1,0\}$, $\{1,3\}$, $\{1,4\}$, $\{2,0\}$, $\{2,3\}$, and $\{2,4\}$, of which five expose the fault, or more than one-half of the possible test sets. Therefore, C_1 subsumes C_2 but the *probability* that a test set selected using C_2 will expose a fault is higher than that for a test set selected by C_1. The reason that this happened was that 0, the only input that failed, occurred in both of C_2's subdomains, but was in only one of C_1's subdomains. Therefore 0 could be selected as the representative of either or both of C_2's subdomains, but as the representative of only one of C_1's subdomains.

3 Using Probabilistic Measures

The observation described at the end of the previous section led Frankl and Weyuker to consider defining relations that would address problems of this nature. Several researchers had previously used a probabilistic measure M to assess the ability of a testing approach to uncover faults and determine whether one criterion was more effective at finding faults than another, including [4, 8, 14]. In Reference [5], Frankl and Weyuker introduced the covers and universally covers relations and used M as a way of assessing whether testing strategies related by these relations were guaranteed to be more effective at detecting faults. M was formally defined for program P whose subdomains are $\{D_1, D_2, ..., D_n\}$, specification S and test selection criterion C as follows: Denoting the size of subdomain D_i by d_i, and letting m_i be the number of failure-causing inputs in D_i, then

$$M(C, P, S) = 1 - \prod_{i=1}^{n} (1 - \frac{m_i}{d_i}).$$

Assuming the independent selection of one test case from each subdomain using a uniform distribution, M is the probability that a test suite will expose at least one fault.

Frankl and Weyuker formally defined the covers relation as follows: Let C_1 and C_2 be criteria, and let $\mathcal{SD}_C(P, S)$ denote the nonempty multiset of subdomains from which test cases are selected to satisfy criterion C for program P and specification S. C_1 *covers* C_2 for *(P,S)* if for every subdomain $D \in \mathcal{SD}_{C2}(P, S)$ there is a collection $\{D_1, \ldots, D_n\}$ of subdomains belonging to $\mathcal{SD}_{C1}(P, S)$ such that $D_1 \cup \ldots \cup D_n = D$. C_1 *universally covers* C_2 if for every program, specification pair (P, S), C_1 covers C_2 for (P, S).

They showed in [5], that a number of well-known criteria are related by the covers relation, but that even if C_1 covers C_2 for (P, S), it is possible for $M(C_1, P, S) < M(C_2, P, S)$. They showed that this sort of inversion happens when a subdomain of the covering criterion is used to cover more than one subdomain of the covered criterion.

This led Frankl and Weyuker to define a new relation, the properly covers relation, in Reference [5]. To solve the problem observed for the covers relation, this new relation requires that this can't happen. Formally we have:

Let $\mathcal{SD}_{C1}(P, S) = \{D_1^1, \ldots, D_m^1\}$, and $\mathcal{SD}_{C2}(P, S) = \{D_1^2, \ldots, D_n^2\}$. C_1 *properly covers* C_2 *for (P,S)* if there is a multi-set

$$\mathcal{M} = \{D_{1,1}^1, \ldots, D_{1,k_1}^1, \ldots, D_{n,1}^1, \ldots, D_{n,k_n}^1\}$$

such that $\mathcal{M}$ is a sub-multi-set of $\mathcal{SD}_{C1}(P, S)$ and

$$\begin{aligned} D_1^2 &= D_{1,1}^1 \cup \ldots \cup D_{1,k_1}^1 \\ &\vdots \\ D_n^2 &= D_{n,1}^1 \cup \ldots \cup D_{n,k_n}^1 \end{aligned}$$

Informally this says that if C_1 properly covers C_2 then each of C_2's subdomains can be "covered" by C_1 subdomains (expressed as a union of some C_1 subdomains). In addition, it must be done in such a way that none of C_1's subdomains occurs more often in the covering than it does in $\mathcal{SD}_{C1}$, thereby preventing the sort of misleading view of the criteria's effectiveness that we saw in the earlier example.

C_1 *universally properly covers* C_2 if for every program P and specification S, C_1 properly covers C_2 for (P, S).

It was proved in [5] that if C_1 properly covers C_2 for program P and specification S, then $M(C_1, P, S) \geq M(C_2, P, S)$.

Thus we can say in a concrete way that criterion C_1 is more effective at finding faults than criterion C_2, for a specific program P. Generalizing this theorem by using the universally properly covers relation they showed that there were many well-known test selection criteria that were related by the universally properly covers relation, and hence these criteria could in a sense be ranked.

In a follow-up paper [6], Frankl and Weyuker next investigated whether there were other appropriate ways of assessing the fault-detecting ability of a criterion, and therefore considered E, the *expected number of failures detected.* Again, letting $\mathcal{SD}_C(P, S) = \{D_1, \ldots D_n\}$, and assuming independent random selection of one test case from each subdomain, using a uniform distribution, E was defined to be:

$$E(C, P, S) = \sum_{i=1}^{n} \frac{m_i}{d_i}.$$

Frankl and Weyuker proved that it is also true that if C_1 properly covers C_2 for program P and specification S, then $E(C_1, P, S) \geq E(C_2, P, S)$.

They also provided examples that showed that in the case in which C_1 subsumes C_2, but *does not* properly cover C_2, this is not necessarily the case.

Thus, by using the universally properly covers relation, Frankl and Weyuker were able to rank testing criteria using both the probability of detecting at least one fault and the expected number of faults exposed, and did so for roughly a dozen well-known testing criteria.

4 Limitations of Formal Analysis

Although the universally properly covers relation is the most natural and promising way of assessing the effectiveness of testing strategies proposed to date, there are still a number of limitations associated with any such analysis that should be recognized.

One important problem that was not considered above is that all of these relations were applied to compare idealized versions of testing strategies which are virtually never used in practice. In a sense, that is the chronic problem associated with formal approaches to software processes. I began this talk by discussing my experience with testers saying that they never saw formal specifications in practice, and therefore, no matter how good a formal approach to testing might appear to be, it was not of interest to them if it was predicated on a formal specification which did not match their reality.

Similar types of issues are associated with all of the comparison relations that I discussed above. For example, one commonly considered testing criterion is *branch testing*, also known as *decision coverage.* Informally, branch testing requires that sufficient test cases be included so that every branch or outcome of a decision statement in the program under test be exercised at least once. No mention is made of how those test cases are to be selected. More formally, in order to prove the sorts of theorems that we've discussed above, it was necessary to make the process more precise. For this reason, the formal definition of branch testing assumes that the domain is first divided into subdomains, each containing exactly those members of the input domain that cause a given branch in the program to be exercised. Then it is assumed that one element of each subdomain is randomly selected using a uniform distribution.

It is difficult to imagine that this process would ever be followed in practice. Pragmatically, branch testing tends to be used more often as a way of assessing the thoroughness of testing, rather than as a basis for selecting test cases. A tester typically selects test cases based on intuition and experience until they believe they have done a comprehensive job. They might then use a branch coverage tool that determines the percentage of the branches of the program that have been exercised by the test suite they've assembled so far. If the percentage is high, the tester might then see which branches had not been covered and try to determine an input that would cause each of the uncovered branches to be exercised. If the percentage was low, then the tester would likely continue to use ad hoc methods of selecting test cases and then reassess the branch coverage achieved, iterating until the percentage of branches covered exceeded a prescribed level or the tester believed that they had done enough.

If branch testing is used in that manner, then we know *nothing whatsoever* about how its effectiveness compares to other testing strategies, because we did not assess *that* version of branch testing, we assessed an entirely different testing method. The same is true for all of the testing strategies compared - they were not the *real* strategies that testers use to select test cases, they were idealized versions.

Another limitation of the work described above is that there is no provision for human variability. Each tester comes to the table with their own set of experiences, expertises, and acquired intuition, and so two different testers using exactly the same approach to test a given software system will generally select different sets of test cases. This is *not* because of the variability due to random selection - it is because of individualized human behavior and the latitude provided in the selection of elements of a subdomain. How can we codify this behavior so that we can say meaningful things about the effectiveness of using different testing methods?

A different kind of issue is related to the appropriateness of relying on the measures M and E as a basis for feeling confident that one testing strategy does a better job than another. Neither of these measures differentiates at all between high consequence faults and trivial faults. Therefore, if criterion C_1, the criterion that properly covers criterion C_2, exposes trivial faults while C_2 exposes catastrophic faults, then the fact that the program was tested using C_1 does not really indicate that it is more dependable than it would have been if it had been tested using C_2. This is true in spite of the fact that more faults were uncovered using C_1, and there was a higher likelihood of exposing faults, since the ones you were exposing were of little consequence.

And what about the cost of doing the testing itself? Perhaps C_1 *does* do a somewhat better job of testing than C_2, and exposes more faults of equivalent severity than C_2 does. But what if C_1 costs orders of magnitude more to use than C_2? Is the added benefit worth the added cost?

Perhaps the biggest problem is that when we are done testing the software using a given criterion, even if it really is the best of the ones being considered, what do we know about the dependability of the tested software? That problem has not been addressed at all by this work. Are there concrete ways that we can determine that information?

5 Comparing Criteria Empirically

So far we have discussed formal analytic ways of comparing software testing criteria. It is also possible to compare these criteria empirically, and, in fact, there are two distinct

sorts of empirical studies that could be performed. The first involves doing a formal scientific experiment, while the second involves a far less formal case study.

Formal experiments generally involve applying the technique under consideration to a substantial population of software systems and observing various characteristics of this application, such as cost, effectiveness, or ease of use. Formal experiments have the advantage that you can extrapolate from the results observed during the experiment to other systems, because the subjects of the experiment were supposedly representative of the larger population which is your universe. But true scientific experiments are rarely if ever done in this area since it would require that there be a clear understanding of what is meant by a "typical" program containing "typical" faults, and we generally do not have that sort of information, even in limited domains such as medical or telecommunications.

In contrast to this, a case study examines the application of a testing strategy or characteristics of testing strategies for one or a few specific systems. Although it is difficult to do them well, and expensive to design and perform them, it is nonetheless generally much more feasible to perform carefully crafted case studies than formal experiments. Unfortunately, however, in most cases it is either difficult or impossible to extrapolate from results observed for the specific systems that served as subjects of the case study to systems in general since the subjects were typically *not* selected because they were especially representative, but rather because there were project personnel who were willing to participate in the study, or management support, or because the project itself initiated the study to find out information about *their* project.

When doing a case study for a large industrial software system, it is often necessary to *model* the system which involves designing a simplified version of the system which is close enough to reality that the observations made about the model are valid for the full system, yet simple enough to be tractable. Our experience has been that most software testers find modeling very difficult to do. In addition, when the case study is completed for the simplified system, you don't necessarily have an accurate picture of how the real system will behave.

In addition to these limitations of case studies, there are general problems that are similar to those associated with the formal analytic comparison relations. For example, case studies may also involve the use of idealized ways of using the criterion. In addition there may not be any provision to account for individualized behavior. If the case study is just intended to assess the effectiveness of different test case selection strategies, then cost may not have been assessed during the case study and no cost-benefit study may have been performed. Similarly, since the fault severity is generally not an integral part of the test case selection criteria, it would be unusual if a case study designed to compare the effectiveness of different strategies would even consider this dimension, and for similar reasons, there is no reason to expect that an assessment of the overall state of the software will have been made at the end of the case study.

But case studies do have some real positive characteristics too. They can provide a "proof of concept" which may be sufficient to encourage practitioners to try the technique being investigated, in the field. It may be considered too big a risk to try a new testing technique or other software development strategy on a large production software project because of time and reliability constraints. However, if it can be shown in a case study that other similar projects have used the technique and gotten better results than the currently-used process, that might be sufficient to convince management of the usefulness of the new approach. Additionally, case studies can provide an estimate of the difficulty of using the test selection criterion. Often it may seem to project personnel that the

learning curve will be too steep to make the adoption of the new technique worthwhile. By showing how other projects have adopted the use of the criterion, and including a description of the experience of testers, it might be possible to convince a project to try using it too.

Another issue that may serve as a roadblock to the adoption of a new test case selection criterion is the perceived or actual cost of using it. Estimates of this cost can be essential in convincing management of the practicality or feasibility of using the strategy. Thus, one test case selection strategy may find 10% more faults than another, but if it costs one hundred times more, or it is *feared* that it will cost much more to use, then it might be dismissed out of hand. A comprehensive case study that includes an accurate assessment of its cost to use on one or more real projects may be sufficient to win acceptance for the strategy.

One other useful potential payoff of doing a large industrial case study to determine the effectiveness of a software test selection criterion relates to modeling. It is often difficult to determine an effective granularity for modeling the software. When it is necessary to model a system, particularly when the system is large, the task can sometimes seem overwhelming to practitioners. In these situations, a case study that includes a description of the level of granularity at which the modeling was done can be extremely helpful.

Our personal experience has been that large industrial case studies are difficult and expensive but very valuable for the reasons mentioned above, and therefore well worth the time and effort. We have performed a number of case studies including those described in [1, 2, 3, 9, 10, 13] and found them to be extremely helpful for the reasons discussed above.

6 Conclusions

We have studied a variety of proposed ways of comparing software testing criteria, and found that many of the formal comparison relations had profound problems associated with them, even though they initially appeared to be intuitively reasonable. We argued that the least problematic comparison relation was the properly covers relation, since it made concrete what it meant for one criterion to do a more effective job of testing than another. However, even this relation was not without serious flaws.

Of central importance was the fact that all of the criteria that were compared by Frankl and Weyuker were idealized versions of the way practitioners actually test software. So, even though we know precisely how these idealized criteria compare based on the expected number of faults detected and the probability of finding faults, we are not able to state conclusively how the pragmatically used versions of these strategies stack up.

Also, none of the proposed comparison relations took the cost into account. If we knew that one criterion was a little better at finding faults than another but that it cost many times as much, would the more effective strategy still be attractive or even feasible to use? That would likely depend heavily on the application and its reliability requirements. And if criterion C_1 was deemed more effective than criterion C_2 by some relation, but all the faults that were being detected by C_1 were trivial faults, but C_2 found profound and potentially catastrophic faults, would we still consider C_1 a more effective criterion? By all the relations we investigated they would be considered stronger, since there is no way to factor in the severity or consequence of the faults.

Perhaps most important is that there is no indication whatsoever of how dependable the software will be when it has been thoroughly tested according to some criterion. Even knowing that criterion C_1 always does a more effective job of testing than criterion C_2, does not imply that C_1 does a *good* job of testing.

So what are the implications of all this? First, this is a call for new thought and more important, new types of thought to be put into attacking this problem. We saw that the initial proposals for comparison relations seemed reasonable at first glance, but turned out to be flawed because they really did not properly capture the essence of what it means for one testing strategy to be more effective at finding faults than another. Then there was the insight that using probabilistic measures would be more appropriate, and in fact they *are* more appropriate. But there are still serious flaws. I do not see a way to address the problem that we are comparing idealized versions of the testing strategies. It is very difficult to imagine how to formalize what is in practice a very individualized process.

I also see a very important role for empirical studies. They can tell us things that all the theorems in the world cannot tell us - how something really works in practice. I see comparing and assessing software testing strategies as a very important and essential problem that must be solved in order to elevate the practice of testing. I think the best hope for a solution will be the combination of carefully thought out theory, done by people who understand both theory and practice, along with carefully thought out case studies. It will involve a great deal of thought, a great deal of ingenuity, and a great deal of effort, but hopefully it will all be worth the trouble.

References

[1] A. Avritzer and E. J. Weyuker. The Automatic generation of load test suites and the assessment of the resulting software, *IEEE Trans. on Software Engineering*, Sept 1995, pp.705-716.

[2] A. Avritzer and E. J. Weyuker. Deriving workloads for performance testing *Software Practice and Experience*, Vol. 26, No.6, June 1996.

[3] A. Avritzer and E. J. Weyuker. Metrics to assess the likelihood of project success based on architecture reviews. *Empirical Software Eng. Journal*, Vol. 4, No. 3, Sept. 1999, pp.197-213.

[4] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(7):438–444, July 1984.

[5] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault detecting ability of testing methods. *IEEE Transactions on Software Engineering*, pages 202–213, Mar. 1993.

[6] P. G. Frankl and E. J. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, Oct. 1993.

[7] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, SE-9(6):686–709, Nov. 1983.

[8] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402 – 1411, Dec. 1990.

[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. Proceedings of the 16th International Conference on Software Engineering, May 1994, pp.191-200.

[10] T.J. Ostrand and E.J. Weyuker. Collecting and categorizing software error data in an industrial environment. *J. Systems and Software*, Vol.4, 1984, pp.289-300.

[11] S. Rapps and E. J. Weyuker. Data flow analysis techniques for program test data selection. In *Proceedings Sixth International Conference on Software Engineering*, pages 272–278, Sept. 1982. Tokyo, Japan.

[12] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-14(4):367–375, Apr. 1985.

[13] E. J. Weyuker and A. Avritzer. A metric to predict software scalability. *Proc. 8th IEEE Symposium on Metrics (METRICS02)*, June 2002, pp.152-158.

[14] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[15] E. J. Weyuker, S. N. Weiss, and D. Hamlet. Comparison of program testing strategies. In *Proceedings Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10. ACM Press, Oct. 1991.

Generating Formal Specifications from Test Information

Position paper for FATES'02

Thomas J. Ostrand
AT&T Labs – Research
180 Park Avenue
Florham Park, NJ 07932
ostrand@research.att.com

Abstract

When experienced testers write test cases for programs, they choose the specific test case values to represent sets of similar inputs that they expect the program to receive. Although testers may start by thinking of individual test cases, or groups of related test cases, gradually their understanding of the program increases to a point where they have much more knowledge of the program's behavior than is expressed in a finite set of test cases. We propose using the Category-Partition Method of writing a test specification to capture this more general knowledge, and make it usable as a formal description of the functionality of a program. A Category-Partition test specification consists of clauses that describe sets of values for input parameters, state variables and results of the program. These clauses fulfill two purposes: they can be compiled by a suitable processor into a set of actual test cases for the program, and they also serve as a description of the program's functionality. The latter purpose is the subject of this paper.

1 Introduction

The usual viewpoint of formal approaches to testing is to derive test cases from a formal specification of the software. This can be difficult to implement in practice because most software does not have a formal specification, and most software engineers and test designers lack training in formal methods.

In this paper we reverse the usual direction, and consider the possibility of generating formal specifications from test cases and more general information available to the tester. Although at first glance this may seem like doing things in the wrong direction, there are many reasons why it can be useful to generate a formal specification after the system has already been built and tested. The specification can be used as the basis for the next version of the system, which could then be built using more disciplined techniques. Once the formal specification is at hand, additional test cases could be derived from it. The specification can be the basis for preparing user documentation, and can be used to explain the system to potential users and customers. Finally, a formal specification can be a basis for analyzing and verifying the system's properties; even after a thorough set of tests has been executed with correct results, it may be discovered that the wrong system has been built.

For many systems, not only formal specifications, but any type of complete and unambiguous specifications are lacking. Many systems are designed only on the basis of high-level requirements, plus the ideas in the designers' heads of what the proper functionality, behavior, and appearance of the working product should be. These ideas are supported by specific instances that illustrate the desired behavior, and the collection of these instances may be as close as the system ever gets to a specification.

How is such a system tested? The first test cases are the specific instances that were used to design the system in the first place. Obviously, if the system is to be even close to correct, it must operate as expected on these cases. As the testers and developers become more familiar with the system, additional tests are developed, perhaps with the cooperation of the system's eventual users. A common method of creating these tests, especially for interactive systems, is by defining scenarios or use cases, which are examples of the system's behavior. Other cases may be created by observation of the existing cases; the system's behavior on a specific case may suggest some

small variations in the inputs or the system state that could affect the test's outcome, and these variations become new test cases.

After the system is placed in operation, a new source of tests arises out of failure reports from the field. After a failure-causing fault has been repaired, the inputs and system state that produced the failure are added to the growing test base. As successive versions of the system are produced, regression testing becomes important, and a common policy is to leave all fault-induced test cases in the regression suite.

We propose that the test cases developed for a system can be the foundation of a formal specification for the system. The expected outputs of tests define key values of the program's input/output relation. Test cases that run successfully can delineate the bounds of the program's valid input space; cases that do not run successfully point out places that are outside the normal space. Tests that run successfully, but are treated differently from the majority of successful inputs, define discontinuities or inflection points of the software.

Any set of test cases specifies part of the behavior of the system, namely the finite set of input-output pairs of the tests. To be more generally useful, however, a finite set of tests must be capable of being generalized to specify behavior that is not explicitly given by one of the test cases. One way of doing this is through program inference [Wey83]. Weyuker proposed using program inference as a means of assessing the adequacy of a set of test cases. This approach suggested that a set of test cases could be considered adequate to test the behavior of a program if the program inferred from the test cases were equivalent to the actual program being tested. Although this is an appealing theoretical framework for characterizing test adequacy, it faces two daunting problems that are in general undecidable, and in practice always difficult to accomplish: inferring a program from samples of its behavior, and determining program equivalence.

However, much can be gained from the test writing process short of a complete characterization of the program. When a tester defines a test case, he is frequently thinking not of the one set of specific values of the test, but rather of a range of values that the single case represents. This is the basis of the equivalence class approach to test generation [GG75, WO80, WJ91], probably the most fundamental concept in testing.

The category-partition test specification [OB88, BHO89] captures exactly this type of more general description of equivalence classes of test data, and is a natural formalism for representing the program's behavior.

2 Converting Informal Requirements into Formal Specifications

Consider the following natural language description of some of the features of a lending library system.

The library has three different classes of members: regular members, junior members, and child members. Child members are ages 3-9, junior members are ages 10-14, and regular members are ages 15 and up. In addition, regular members who meet certain requirements can apply for a privileged membership.

Each book in the library is classified as a regular book, a reference book, or a restricted book. Reference books do not circulate, and restricted books either circulate for a shorter than normal time period, or can be borrowed only by a privileged member. To encourage young people to read, a child or junior member may take out books for longer than the normal time as long as they do not owe any fines.

If borrowed books are returned past their due date, the library assesses a fine on the borrower of .25 per day. Borrowers owing more than $5.00 are considered major delinquents, and are not permitted to borrow any further books until their account is settled. No member is allowed to borrow more than 10 books at any one time.

Software is to be written that will manage the needed functions of the library. Some of the required operations that handle the interactions of users with the library are Join and Leave the Library, Borrow and Return a Book, Reserve a Book, and Send Bill. Other operations pertain to managing the library collection, such as Enter New Book, Retire Existing Book, Classify Book.

As with many requirements documents, this description leaves many questions unanswered. What are the requirements for privileged membership? What is the basis for classifying a book? What is the circulation time for normal books and for restricted books? If a privileged member borrows a restricted book, does it circulate for the normal time, or for the shorter restricted time? Does the 10-book maximum apply to each instance of visiting the library to borrow books, or does it mean that the total number of books currently being borrowed by a member cannot exceed 10?

Obviously these and many other questions must be answered before the software can be successfully released; the system architect, the designer, or possibly the developers must seek the answers from the customers. Further, the developed system obviously can't be tested or validated without the answers, since there would be no way of knowing whether the system is doing what it's supposed to do.

The ambiguities of informal requirements are brought to the surface by attempts to formalize the requirements. Instead of asking software engineers to use a formal specification language like Z or VDM, we propose the use of the test specification language of the Category-Partition test design method (CPM). With CPM [OB88, BHO89, AO94, Bin99], the tester analyzes the available information about a function, and writes information describing the function's inputs, the environment in which it operates, and the results that the function produces. The written information constitutes a test specification, which can be processed to produce test cases for the implemented software.

The CPM language is quite simple to learn, and testers are already familiar with the activity of designing test cases. Over the past 24 years, the author has been associated with three major software-producing companies. The goal has always been to produce correct, efficient software that supports the company's products. To the best of the author's knowledge, formal specification techniques have not been used in any of their software development processes, but there have always been dedicated testing organizations in place. While the testers are usually quite good at their jobs, and usually employ systematic approaches and current automation tools, they always have a desire for a more structured and more complete approach. To meet this need, the category-partition method was introduced at one of these companies, and was employed successfully in several projects [OW92], proving its viability in a commercial software development environment.

3 The Category-Partition Method

A category-partition test specification is in effect a program written in a Specification Language for Test Generation. The output produced by executing the test specification program is a set of test cases for the function. The specification contains two main parts, the *Input Section* and the *Result Section*; they are described below informally. There is presently no compiler available for the test specification language, although early versions have been written and used [BHO89, OW92]. To illustrate the method, we develop a test specification for the Borrow-Book function of the library system. At various points where the given requirements are not specific, we shall make assumptions about the proper behavior of the system.

The Input Section

Creating the input section starts with identifying the distinct attributes, called *categories*, of the input and state space that affect the behavior of each function. After a category has been identified, the tester partitions it into disjoint sets of values, called *classes*; the goal is that each class should represent a set of values that have the same effect on the function's behavior, so each value in the class should be equivalent for testing purposes.

In general, inputs and state conditions can affect a function's behavior individually or in combination with each other. After the categories have been partitioned, the tester then specifies how the different classes interact when the function executes: the tester specifies valid and invalid input and state combinations, and the resulting outputs and state changes that occur with each combination.

The information produced as a result of this analysis can be used to define test cases that include all combinations of the input and state condition classes. The test cases can be produced manually from the category-partition information, but it is far more productive and effective to use an automated tool. When a tool is available, the information is written in a structured test specification that provides a blueprint for how the values in the categories are combined to form test cases for the function.

Suppose the Borrow-Book function has two inputs, a *book-id* and a *member-id*. The book-id has at least two categories: the book's type and its status in the library. The member-id has at least three categories: the member type, the member's delinquency status, and the number of books the member is currently borrowing. We start with the outline of a test specification for these two inputs, giving only logical names for the categories and the classes within each category.

Input Section for Borrow-Book

Book-id
- Book-type
  - invalid
  - regular book
  - restricted book
  - reference book
- Status
  - book checked out
  - book available
  - book non-circulating

Member-id
- Member-type
  - invalid
  - regular
  - junior
  - child
  - privileged
- Delinquency
  - none
  - minor
  - major
- Number-out (number of books currently borrowed)
  - none
  - at least 1, less than maximum
  - maximum

Input section categories can be static or changeable. *Book-type* and *Member-type* are static attributes; for a given book or member, their values do not change. *Status*, *Delinquency*, and *Number-out* are changeable, state-based attributes; their values are maintained by the system and may change as a result of executing the function. The state-based attributes can appear again as categories in the Result section of the test specification, with conditions that describe how they change as a result of executing the function.

The cross product of the input categories defines the entire potential input space for test cases. For the Borrow-Book specification, there are 4x3x5x3x3 = 540 possible input combinations. The specification language permits classes to be annotated with *restrictions* or *limits* that reduce the number of combinations that are produced. For example, the tester may decide that an invalid Book-type will be sufficiently tested with a single member-id, rather than the 45 (= 5x3x3) combinations that are defined in the specification. This could be accomplished by placing the restriction

[Member-type = regular & Delinquency = none & Number-out = none]

on the invalid class of Book-type. As a result, three tests would be generated, corresponding to the three classes of the Status category. The annotations are described in more detail in [BHO89].

To generate actual test inputs from the Input Section, each class is annotated with a range of values. For example, the Delinquency category's classes could be specified as follows:

Delinquency
    none (\$0.00)
    minor (\$0.01 .. \$4.99)
    major (\$5.00 ..)

The notation implies that a suitable value for the *none* class is \$0.00, for *minor* anything between \$0.01 and \$4.99, and for *major* \$5.00 and greater. The test specification processor would generate specific input values using these ranges. Similar range information could be supplied for the other input categories. The Book-type and Member-type ranges would depend on the particular format of the id's that are used by the library. The Book-id Status is a Boolean value for each class, and Number-out has the ranges (0) for *none*, (1 .. k) for *at least 1*, and (k) for *maximum*. According to the informal specification, the value of k is 10, but the tester might want to vary k to test the program more thoroughly.

The Result Section

Results describe the outputs and state changes produced by the function; they are categorized and partitioned just like the inputs. The tester must decide what characterizes the function's results, and then come up with categories that capture the function's behavior. For the Borrow-Book function, the key result is the actual Borrowing: can the member take the book out; if so, for what period of time; if a book is borrowed, how is the member's information updated? These aspects of the Borrow result are captured in the categories *Borrow-permission*, *Length-of-loan*, and *Number-out*. Borrow-permission is a Boolean value that states whether the Borrow is permitted. Length-of-loan is a numeric value giving the number of days the book may be kept by the member. If the member is permitted to borrow the book, the number of books presently out for the member increases by 1, represented by the *Number-out* category. The Result category Number-out represents the same state-based attribute as the Input category Number-out. The Result category will later be annotated with conditions that describe how the category changes in terms of the input values.

Besides managing the details of the Borrow operation, the system might also generate a notice for the borrower, describing the result of his transaction with the library. This notice output is captured in the *Notice-to-Member* result, with the single category *Borrow-notice*, which is partitioned into 4 classes that characterize the result of the Borrow attempt.

Result Section for Borrow-Book

Borrowing
    Borrow-permission
        yes
        no
    Length-of-loan
        regular period
        short period
        long period
    Number-out
        none
        $1 \leq k < \text{maximum}$
        maximum
Notice-to-member
    Borrow-notice
        Borrow ok; date due
        Book out; date due
        Can't borrow: reference book
        Can't borrow: owe more than \$5

The format of the category-partition specification permits easy modification when it becomes necessary to test for other characteristics of the output. For example, a later version of the software might deliver to the library user a memo listing upcoming events at the library, such as speakers, musical programs, films, or special book sales. This could be easily grafted into the *Notice-to-member* result by adding a new category *Events notice*, and partitioning it into the classes mentioned.

After determining the Results and their categories, the next step is to write expressions that characterize the conditions under which each class of a result partition will occur. These expressions are written in terms of the categories and classes of the input parameters, and will become the basis for creating the oracle part of the test cases. Below, we show the result part of the test specification, with the addition of the conditional expressions. Conditions[1] can be attached to both the category and to the classes within a category. A category condition must be satisfied by every class of the category, while a class condition is satisfied only by the individual class.

Borrowing
- Borrow-Permission
  - yes [id ≠ invalid & book-status = available & delinquency ≠ major & number-out < max & (book-type = regular | (book-type = restricted & id = privileged))]
  - no [id = invalid | book-status = out | delinquency = major | number-out ≥ max | book-type = reference | (book-type = restricted & id ≠ privileged)]

- Length-of-loan [Borrow-Permission = yes]
  - regular period [delinquency ≠ major]
  - short period [book-type = restricted & id = privileged]
  - long period [(id = child | id = junior) & book-type = regular & delinquency = none]
- Number-out
  - none [Number-out = 0 & Borrow-permission = no]
  - 1 ≤ k < max [(Number-out = k-1 & Borrow-permission = yes) | (Number-out = k & Borrow-permission = no)]
  - maximum [(Number-out = maximum-1 & Borrow-permission = yes) | (Number-out = maximum & Borrow-permission = no)]

Notice-to-member
- Borrow-notice
  - Borrow ok; date due [Borrow-permission = yes]
  - Book out; date due [Book-status= checked-out]
  - Can't borrow: reference book [Book-type = reference]
  - Can't borrow: owe more than $5 [Delinquency = major]

The Length-of-loan category illustrates the use of conditions attached to both a category and its classes. Any of the classes in the Length-of-loan category are produced only if the Borrow-permission result is yes, which implies that the user has a proper id, the book is available, and the user is not delinquent. The conditions on the individual classes are further restrictions that define when those specific results occur.

The use of the result category Borrow-permission in the Length-of-loan and Number-out conditions is like a macro call that will be replaced by the atomic conditions in the Borrow-permission classes.

The CP specification is created by a human, based on the tester's knowledge of the system, and experience in designing test cases. As with any human-created artifact, it is subject to errors of omission and commission. In particular, with a complex system, it may be difficult to assure that the result conditions are consistent and complete. A specifier might, for instance, write

[1] Conditions are surrounded by square brackets [ ]. "&" represents logical AND. "|" represents logical OR.

non-disjoint conditions for two different classes of a category, implying that some specific input combination produces two different results. If this happened, then any test case that satisfied the conditions for both classes would be expected to produce both results. Running an actual test case that satisfies the two conditions would reveal the discrepancy.

Completeness is a more difficult problem. Just as a major issue of testing is deciding whether a set of test cases provides "complete coverage" of the functionality to be tested, there is no general way to know if a CP specification is a complete description of the function. The tester could omit a disjunct or conjunct from a condition, with the result that the test cases are under-constrained. Omitting the *id* $\neq$ *invalid* element from the Borrow-permission *yes* class could yield test cases with an invalid member-id allowed to borrow a book.

4 Test Specification as Formal Specification

The CP specification above has all the elements of a standard formal specification. The Input Section categories describe the function's inputs, and delimit the value ranges of the inputs. The Result section describes the function's possible outputs; the conditions on an individual class of a category are pre-conditions for performing the operation to yield a value within that class. The classes themselves act as post-conditions that specify the value of the category after performing the operation. Thus,

[Length-of-loan = Short-period]

is a post-condition that should hold after Borrow-book is executed with the pre-conditions

[Borrow-permission = yes & Book-type = Restricted & Member-id = privileged].

The CPM supplies two benefits when it is used to help generate a formal specification. First, by focusing on the test cases, the specifier has concrete values in mind, enabling clear descriptions of the input categories, and the conditions that lead to the results. Test execution provides more opportunity to arrive at an accurate and correct formal specification. Although the usual assumption is that the test specification is a correct description of the desired functionality, and that a discrepancy between the expected result (the test case) and the actual result (the implementation) indicates a mistake in the implementation, this may not be the case. Test results may indicate a need to change the specification to bring it in agreement with the implementation.

The second benefit is the CPM's "divide-and-conquer" approach of analyzing the individual parameters and state variables separately. In actual use, we have found this approach makes it easier to write the specification, and is an aid to understanding the overal behavior of the function.

5 Summary

The proposal of this paper is to use test cases and test specifications as a means of arriving at a formal specification of a program's functionality. Writing test cases is frequently the first action of a programmer after software has been implemented; in fact, a good programmer may write the test cases before the code. A thorough set of test cases should characterize the program's functionality well enough to allow a reader to determine the program's behavior in additional cases. This is, of course, exactly what is achieved by a formal specification.

We have used the category-partition formalism to express test cases, and shown how this formalism can resemble a traditional formal specification. Many issues remain open. Major problems include determining whether the test specification is complete and consistent.

References

[AO94] P. Ammann and J. Offutt, Using formal methods to derive test frames in category-partition testing, *Proc of COMPASS 94*, Gaithersburg, MD, June 1994, 69-80.

[BHO89] M. Balcer, W. Hasling, T. Ostrand, Automatic generation of test scripts from formal test specifications, *Proc of SIGSOFT 89, Third Symp. Software Testing, Analysis, and Verification*, Dec 1989, 210-218.

[Bin99] R.V. Binder, *Testing Object-Oriented Systems*, Addison-Wesley, Reading, MA 1999, 419-426.

[OB88] T. Ostrand and M. Balcer, The Category-partition method, *Communications of the ACM*, Vol 31, No 6, June 1988, 676-686.

[Wey83] E. Weyuker, Assessing test data adequacy through program inference, *ACM Transactions on Programming Languages and Systems*, Vol 5, No 4, Oct 1983, 641-655.

[WJ91] E. Weyuker and B. Jeng, Analyzing partition testing strategies, *IEEE Transactions on Software Engineering*, Vol 17, No 7, Jul 1991, 703-711.

[OW92] T. Ostrand and J. Wood, Industrial applications of the category-partition method, *Proc Pacific NW Software Quality Conference*, Portland, OR, Oct 1992.

[GG75] J.B. Goodenough and S.L. Gerhart, Toward a theory of test data selection, *IEEE Transactions on Software Engineering*, Vol 1, No 2, June 1975.

[WO80] E.J. Weyuker and T.J. Ostrand, Theories of program testing and the application of revealing subdomains, *IEEE Transactions on Software Engineering*, Vol 6, No 3, May 1980, 236-246.

Testing from statecharts using the Wp method

K.Bogdanov, M.Holcombe

emails: {K.Bogdanov, M.Holcombe}@dcs.shef.ac.uk
tel: +44(0)114 2221847, fax: +44(0)114 2221810
Department of Computer Science, The University of Sheffield
Regent Court, 211 Portobello St., Sheffield S1 4DP, UK

Abstract

An existing testing method for Harel statecharts with hierarchy and concurrency is based on what is known as the Chow's W method. This paper presents an extension of this statechart testing method to build on the Wp method, making a test set smaller. Subject to some specific conditions and subsequent testing not revealing faults, both the original and the extended testing methods make it possible to prove the correct behaviour of an implementation of a system to its specification.
Keywords: specification-based testing, formal methods, software testing, finite-state machines, statecharts.

1 Introduction

At present, a variety of methods for test generation from state-based models of software are available. Most of the methods perform a coverage of varying kind [18, 2, 25], generate tests from a test purpose [23, 21, 3] or a conformance relation [29, 26, 11]. Such tests can be effective at finding faults but they do not make it possible to decide when to stop testing. Finite-state machine based testing methods [9, 12] consider behavioral equivalence of state machines as a conformance relation and produce a finite test suite to check it. The amount of testing depends on an a-priori known upper bound on the number of states in an implementation automaton. The inability of finite-state machines to represent data without a state explosion can be solved by using functions (further called *labels*) on transitions, which can access and modify global data. This has given rise to notations such as X-machines (also known as extended finite-state machines). Finite-state machine testing methods have been adapted to X-machines by treating labels on transitions symbolically during test generation, i.e. by generating test cases in terms of sequences of labels and then converting them to sequences of inputs to attempt to drive an automaton of an implementation through these sequences [17, 19]. Further, the X-machine testing method has been adapted to test hierarchical and concurrent Harel statecharts [5], where a test suite is built by incrementally following the structure of a specification. Both X-machine and statechart testing methods preserve the complete test guarantee of the original finite-state machine testing method, but introduce a number of restrictions X-machine and statechart specifications and implementations have to comply with. The original statechart testing method was ultimately based on the Chow's W method [9]; this paper describes what needs changing in it to make use of the Wp method [12]. The advantage is a reduction in the size of a test set without weakening of the results obtained by testing. Statecharts are introduced in Sect. 2,

followed by the description of the original testing method for statecharts in Sect. 3. The extension of it is given in Sect. 4; concluding remarks can be found in Sect. 5.

2 Statecharts

Statecharts [13, 14] is a specification and design language derived from finite-state machines by extending them with arbitrarily complex functions on transitions, state hierarchy and concurrency. Consider a simple tape recorder capable of playback, rewinding, fast forwarding and recording as well as changing a side of a tape when the button *play* is pressed during playback or when a tape ends. The statechart to be presented models the control portion of this tape recorder, interpreting user's button presses and sending appropriate commands to a tape drive mechanics. The inputs to this controller are events *play*, *stop*, *rec*, *rew*, *ff* and *tape_end*. Most of them have intuitive meanings; the last event is issued by the tape mechanism when a tape stops. Output variables are *operation* and *ff_direction*, giving a command and a tape direction to the tape mechanism respectively. The *underline* font is used to denote input and output variables and events (variables with a special property described in Sect. 2.2), transition labels are given in *italics* and state names are CAPITALISED.

2.1 State hierarchy

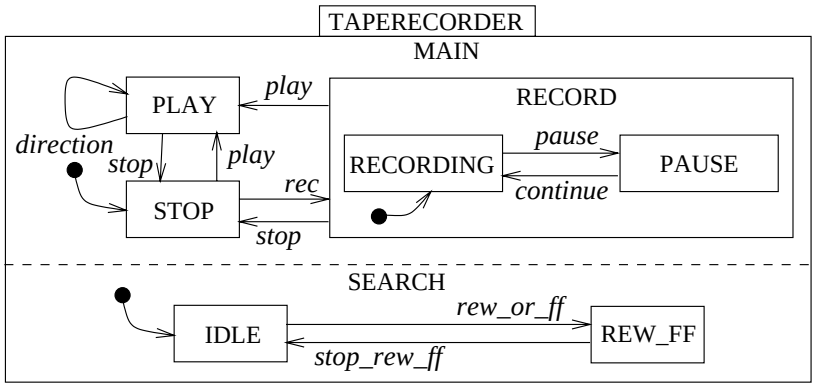


Figure 1: The tape recorder statechart

The statechart is shown in Fig. 1. It consists of two parts running concurrently, MAIN and SEARCH. The first one describes the playback and recording behaviour of the tape recorder and the second one is responsible for forward advance and rewind, making it possible to skip portions of music during playback. Concurrent parts are separated by a dashed line. The TAPERECORDER state containing them is called an AND state. Every concurrent part of it behaves similarly to a finite-state machine with functions on transitions except that some states can have a behaviour defined in them. For instance, the behaviour of the RECORD state is a two-state machine describing whether a tape recorder is actively recording or waiting for a user command. Only one state in any non-concurrent state can be active. While a statechart is idle in the MAIN state, this could be state STOP; during recording this is RECORD in MAIN and RECORDING in RECORD. Non-concurrent states with behaviour in them are called OR states and those without any, such as PLAY and RECORDING — BASIC ones.

Initial states in any OR state of a statechart are pointed at by transitions from blobs. For the MAIN state this is STOP, for RECORD and SEARCH — RECORDING and IDLE respectively. There is no need to do this for AND states as every substate of TAPERECORDER has to be entered whenever TAPERECORDER is. When a transition to an OR-state *s* is taken, it is followed by a transition from a blob in that state. If the initial state of *s* is an OR one, a further transition will be taken; entering an AND-state involves entering the initial state of every concurrent part of it. Consequently, taking a single transition in a statechart typically involves following it with a number of further transitions. The whole set of transitions taken is called a *full compound transition*, abbreviated FCT; the abovementioned blobs are *default connectors* and transitions from them — *default transitions*.

In the example transition labels are named to reflect user actions, i.e., *play* occurs when a user presses the *play* button, *rew_or_ff* occurs if either *rew* or *ff* buttons are pressed. To simplify the presentation, details of transitions' behaviour are not shown on the diagram. A precondition which has to be satisfied for a label to be able to fire is called a *trigger*; an operation carried out by a label on a transition when that transition executes is called an *action*. A transition with a triggered label may only occur when a statechart is in its source state; such transitions are referred to as *enabled*. The *direction* transition is triggered by the *play* button to change the side of a tape and by the *tape_end* event when the side being played back ends. If the MAIN statechart is in the STOP state and a user presses *play*, then both *play* and *direction* labels will be triggered, but only the *play* transition from STOP will be enabled and thus taken by the statechart.

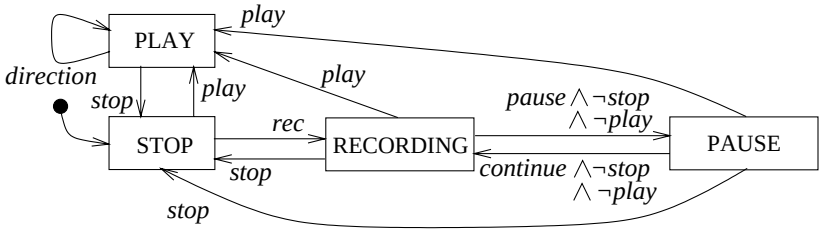


Figure 2: The flattened statechart of the MAIN state

A statechart within a state is left when a transition from that state is taken. For example, the RECORD state is left when the controller takes the *stop* transition, regardless of the substate, RECORDING or PAUSE, it was in. The equivalent statechart to the MAIN state in Fig. 1 is shown in Fig. 2 where the state hierarchy is removed. To do that, the state RECORD has to be replaced by its contents; the outgoing transitions *stop*, *play* and the incoming *rec* one have to be replaced by the five corresponding transitions. The hierarchy of states imposes priorities on transitions in that those at a higher level have priority over lower-level ones; to retain these priorities, labels of transitions between RECORDING and PAUSE states have been appropriately modified in Fig 2.

Full statecharts [14] contain considerably more constructs than those introduced above. Specifically, there could be multiple transitions from default connectors with labels on them while this paper considers a simplified problem where there is exactly one non-interlevel transition from every default connector with no label. Taking labelled default transitions into account appeared to make testing significantly more complex [5]. A variety of connectors, connecting parts of transitions are considered syntactic sugar and are thus not elaborated upon; history can be represented by default transitions to every state and test generation for it is left for future work.

State hierarchy of a statechart can be viewed as a tree; the one for the tape recorder (Fig. 1) is shown in Fig. 3. The *root* state is the implicit top-level state; it was introduced because TAPERECORDER is an AND-state and statecharts require the top-level state to be an OR one [14]. The parent-child relationship between states in the tree is given using the ρ function, similar to [24]. ρ provides a set of substates of a given state. $\rho(\text{RECORD}) = \{\text{RECORDING}, \text{PAUSE}\}$. An opposite to ρ is *parent*, such that $\mathit{parent}(\text{RECORDING}) = \text{RECORD}$. The *scope* of a transition is the lowest-level OR-state above all source and target states of it. For example, the scope of all transitions with labels *play* is MAIN and the one with *pause* — RECORD.

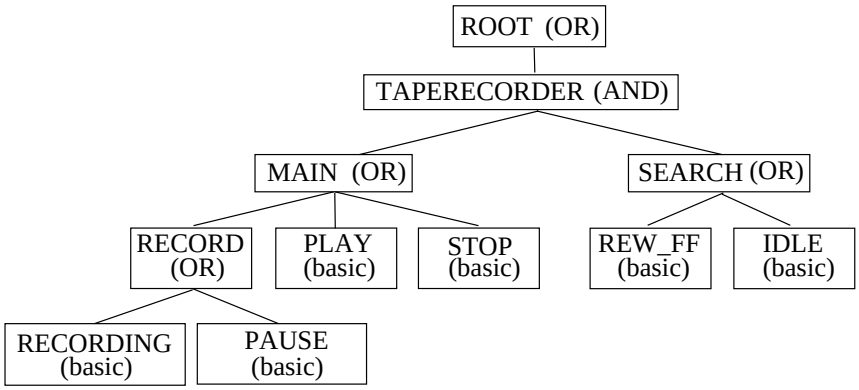


Figure 3: The state tree of the tape recorder

Sets of states which are left and entered by full compound transitions are called *configurations* and consist of states a statechart can be in simultaneously. For example, if the statechart enters the PAUSE state, RECORD should also be entered since it is a parent of the PAUSE one in the state hierarchy. Additionally, if an OR-state is entered, exactly one of its substates must be entered too, for instance, the controller cannot be in the RECORDING and PAUSE states at the same time. Every substate of an entered AND-state has to be entered, so that a possible configuration in Fig. 3 is $\{$*root*, TAPERECORDER, MAIN, SEARCH, RECORD, PAUSE, REW_FF $\}$. A configuration is uniquely determined by a set of basic states in it [24, 5]. Every state in a flattened statechart corresponds to a configuration in the original one. The formal definition of a configuration is given in [24, 7, 5].

Static reactions are a special case of transitions which may occur within a state, without leaving it or entering it again (thus no states are left and no default transitions fire when static reactions are taken). Interlevel transitions are transitions which cross levels of hierarchy. For instance, if the controller had a transition from PAUSE to STOP, it would be interlevel. Interlevel transitions are not considered in this paper. Sequences of labels of transitions (not necessarily those which could be taken) are called *paths* in this paper.

2.2 Step semantics

Assume that an environment the statechart is running in generates some events or changes variables which enable transitions. Transitions which reside in concurrent states, such as *rew_or_ff* and *play*, can be taken together, but not *stop* and *play*. Such a decision is based on whether the lowest common ancestor of scopes of two transitions is an AND-state [7, 27]. Priorities of transitions mentioned above are formally related to their scopes in that a transition with a higher scope w.r.t state hierarchy has a higher precedence than any transition

with a lower scope. Among the enabled transitions which cannot be taken together, one can eliminate from consideration those which have a lower priority to any enabled one. If scopes of any two enabled transitions are the same (such as for *play* and *rec*), there is no rule to prefer one over another one, which implies a nondeterministic choice. In the paper it is assumed that such a situation never occurs in both a specification and an implementation. The set of transitions resulting from elimination is taken by the system. Enabled static reactions in states which were not left or entered by transitions in this set will also be executed. The execution of transitions and static reactions selected as described above, is called a *step*. Transitions taken may in turn generate events and make changes to variables. All changes, including those by the environment, are collected during a step and applied after the step has ended; all events active in the step which were not generated again are discarded. The possible loss of value is what differentiates an event from an ordinary variable. Statecharts considered also have to satisfy the condition that no pair of transitions or static reactions taken in a step modifies the same variable. This implies that actions of transitions and static reactions taken in a step can be executed in any order and justifies the word 'set' used above to describe a collection of them.

Unlike the synchronous time semantics just described, the asynchronous one allows a statechart to perform more than one step in response to actions of an environment, within the same instant of time. This is accomplished by taking steps until no transition or static reaction is enabled. Since this behaviour severely limits observability and controllability of a statechart under test (Sect. 3.4), it is prohibited for a specification and an implementation during testing. For the same reason transitions from states are not allowed to have labels with empty triggers, i.e. those which are always triggered.

3 Test generation for statecharts

In this section the X-machine testing method and its application to statecharts with state hierarchy and concurrency are described following [5, 6, 8]. Initially, the method is given for flat statecharts, followed by the description of testing for hierarchical and concurrent ones. The testing method is primarily aimed at testing an implementation against a detailed specification or a design, although it could be used with minor changes to test from a relatively abstract specification [6].

3.1 Test case generation for statecharts without state hierarchy

With restrictions introduced in Sect. 2, statecharts which do not contain state hierarchy or concurrency are behaviourally-equivalent to X-machines [5] so that the X-machine testing method [19, 17] can be used for testing them. The method is founded on the Chow's W method [9] and relies on a separation of function and transition diagram testing (similar ideas are mentioned in [4]). The method concentrates on testing of the transition diagram; behaviour of the labels of transitions is assumed to have been tested in advance, for example, using the disjunctive normal form (DNF) approach [10, 15]. As a result of testing not revealing faults, an implementation is proven to be behaviourally-equivalent to its specification [19]. The approach to testing of a transition diagram is very similar to testing of labelled-transition systems [28]. The main difference is the reliance of this work on an input/output behaviour of transitions rather than on deadlocks to tell a tester whether a transition with a given label exists from a particular state in an implementation or not. This is addressed in more detail in Sect. 3.4.

For a systematic construction of a set of test cases, auxiliary sets have to be built. Here the MAIN state is used for an illustration, without consideration of the structure of

the RECORD state. The set of transition labels (denoted by Φ) is the set of labels of a statechart, $\Phi = \{stop, play, rec, direction\}$. State cover (denoted by C) is a set of sequences of transition labels, such that one can find an element from this set to reach any desired state starting from the initial one, $C = \{1, play, rec\}$. Here *1* denotes an empty sequence of labels. A characterisation set (denoted by W) allows a tester to check the state arrived at when a transition fires. For every pair of states, it is possible to construct a path which exists from one of them and not from the other. Such paths for every pair of states comprise a characterisation set, $W = \{stop, play\}$. Each element of this particular W is a sequence consisting of a single label. For C and W to exist, a specification of a system has to contain no states having the same behaviour as some others or states with no transitions leading to. This property is referred to as *minimality*.

According to the method, every state has to be entered using C and verified via W. Additionally, every label has to be attempted from every state and in case a transition corresponding to that label fires, the entered state has to be checked. For instance, in order to test the *play* transition from the state RECORD to PLAY, one should begin by entering RECORD from the initial state STOP. This can be accomplished by generating event *rec*; in response, *operation* should change to *rec*, assuming that it means a command to a tape mechanism to start recording. Afterwards, label *play* has to be attempted by generating *play* and observing *operation* changing to *play*. Finally, one needs to test that the PLAY state was entered. This can be done by generating the *tape_end* event to trigger *direction* because transition *direction* exists only from the PLAY state. After triggering it, the modification of the *ff_direction* variable has to be observed. In addition to testing *play* between those two states, it is necessary to test its existence between STOP and PLAY as well as to make sure that no transition labelled by it exists from any other state. The latter test is needed because in a faulty implementation a transition labelled *play* could exist from some state other than RECORD and STOP. The described testing approach yields a set of test cases $C * W \cup C * \Phi * W$ using the notation $A * B$ to denote set multiplication. For some sets of sequences A and B, $A * B = \{a\,b \mid a \in A, b \in B\}$, where $a\,b$ is a concatenation of sequences a and b. $*$ has a higher precedence than set operations $\cup$, $\cap$ and $\setminus$.

For implementations potentially containing more states than the corresponding specifications, longer sequences of transitions have to be tried from every state in order to exercise extra states. Let n be the number of states in a specification and m — the estimated maximal number of states in an implementation. Assuming the possibility of $(m - n)$ extra states, the set of test cases following [17] is

$$T \quad = \quad C * (\{1\} \cup \Phi \cup \Phi^2 \cup \ldots \cup \Phi^{m-n+1}) * W. \tag{1}$$

3.2 Test case generation for state hierarchy

The most simple approach to testing state hierarchy is to flatten a statechart, i.e. turn it into a behaviourally-equivalent one without AND or OR states. For example, Fig. 2 depicts a result of flattening of the MAIN state in Fig. 1. Such a transformation results in a simple but, in practice, huge statechart. Flattening is essentially what is done by [22]. As an alternative, an approach of an incremental test case development using the hierarchical structure of statecharts is proposed. It has the advantage of following the development process and thus the set of test cases can be continuously updated to reflect specification changes made, avoiding the 'big bang' in test case generation. Moreover, if certain parts of a statechart are implemented separately and do not share any labels, one does not have to test for faults where labels from one part are used in another one and vise-versa, significantly reducing the size of a test set [8, 5].

Test case generation begins with the construction of a tuple (Φ, C, W), called a *test case basis* (abbreviated TCB) for every non-basic state considering all its substates as basic ones. Afterwards, one has to walk the state hierarchy bottom-up, merging TCB tuples at every level. The idea of merging is to produce a TCB which could be generated from a flattened statechart. The result of merging for the top-level *root* state $(\Phi^M_{root}, C^M_{root}, W^M_{root})$ can thus be used to generate a set of test cases for the whole system following Eqn. 1. When transitions are added, removed or substates are removed from a state in the process of development, only its own TCB has to be recomputed and merged with TCBs of the higher-level states; addition of non-basic states additionally requires computation of merged TCBs for them.

The elements of the test case basis for the RECORD state are given by $\Phi_{RECORD} = \{pause, continue\}, C_{RECORD} = \{1, pause\}, W_{RECORD} = \{pause\}$. TCB for the MAIN state is $\Phi_{MAIN} = \{play, stop, direction, rec\}, C_{MAIN} = \{1, play, rec\}, W_{MAIN} = \{stop, play\}$. The rule for C constructs paths to all configurations in MAIN, i.e. for every basic state in it, C should have a path leading to it. C_{MAIN} contains paths for basic states directly underneath MAIN and C^M_{RECORD} — for all basic states underneath RECORD, starting from the boundary of RECORD. Consequently, taking all sequences of C_{MAIN} and prefixing all those from C^M_{RECORD} with a path in C_{MAIN} to enter RECORD yields the expected $C^M_{MAIN} = C_{MAIN} \cup \{\text{path in } C_{MAIN} \text{ to enter RECORD}\} * C^M_{RECORD} = \{1, play, rec\} \cup \{rec\} * \{1, pause\} = \{1, play, rec, rec\,pause\}$. W_{MAIN} distinguishes between any pair of states in MAIN and W^M_{RECORD} — between states in RECORD; uniting the two sets gives $W^M_{MAIN} = W_{MAIN} \cup W^M_{RECORD} = \{stop, play, pause\}$, identifying all configurations in MAIN. $\Phi^M_{MAIN} = \Phi_{MAIN} \cup \Phi^M_{RECORD} = \{play, stop, direction, rec, pause, continue\}$ is clearly a set of all the labels in MAIN and its substates.

3.3 Test case generation for concurrency

Testing of concurrency follows the same approach as testing of state hierarchy, except that multiple transitions are attempted. The elements of the test case basis for the top component, MAIN, have been constructed above; those for the bottom one are built similarly: $\Phi^M_{SEARCH} = \{rew\_or\_ff, stop\_rew\_ff\}, C^M_{SEARCH} = \{1, rew\_or\_ff\}, W^M_{SEARCH} = \{rew\_or\_ff\}$. In order to visit all configurations and consider all transitions as well as their combinations, sets C have to be multiplied and W sets be united; Φ^M has to contain a label of every possible set of transitions with orthogonal scopes [7].

$$
\begin{aligned}
\Phi^M_{TAPERECORDER} &= (\{1\} \cup \Phi^M_{MAIN}) \underline{*} (\{1\} \cup \Phi^M_{SEARCH}) \setminus \{1\} = \{play, stop, direction, rec,\\
&\quad pause, continue, rew\_or\_ff, stop\_rew\_ff, play\text{-}rew\_or\_ff, stop\text{-}rew\_or\_ff,\\
&\quad direction\text{-}rew\_or\_ff, rec\text{-}rew\_or\_ff, pause\text{-}rew\_or\_ff, continue\text{-}rew\_or\_ff,\\
&\quad play\text{-}stop\_rew\_ff, stop\text{-}stop\_rew\_ff, direction\text{-}stop\_rew\_ff,\\
&\quad rec\text{-}stop\_rew\_ff, pause\text{-}stop\_rew\_ff, continue\text{-}stop\_rew\_ff\},\\
C^M_{TAPERECORDER} &= C^M_{MAIN} \underline{*} C^M_{SEARCH} = \{1, play, rec, rec\,pause, rew\_or\_ff, play\text{-}rew\_or\_ff,\\
&\quad rec\text{-}rew\_or\_ff, rec\text{-}rew\_or\_ff\,pause\},\\
W^M_{TAPERECORDER} &= W^M_{MAIN} \cup W^M_{SEARCH} = \{stop, play, pause, rew\_or\_ff\}.
\end{aligned}
$$

$A \underline{*} B = \{a \diamond b \mid a \in A, b \in B\}$ where $a \diamond b$ means that sequences a and b are taken side-by-side with ith element of a and b taken in the same step. Notation-wise, in order to take several transitions in the same step, these sets of sequences have to be considered to be sets of sequences of sets with elements of inner sets shown delimited with a dash (-). For example, $(pause\,stop) \diamond rew\_or\_ff = pause\text{-}rew\_or\_ff\,stop$, $(pause\,stop) \diamond 1 = pause\,stop$. Here dash means that *pause* and *rew_or_ff* are taken in the same step, while *stop* (separated by a space) — in the one after it. Multiplication $\underline{*}$ is used for C construction in order to produce

the shortest possible sequence of transitions by taking as many of transitions as possible in the same step. A sequential multiplication $*$ could be used instead, leading to longer test sequences. For the tape recorder under the assumption of an implementation containing no more states than the specification, test case generation produces 672 sequences. It is later contrasted with the size obtained using the Wp method.

Static reactions can be transformed into ordinary transitions through a well-known transformation of a statechart. The idea is to consider static reactions as ordinary transitions executing concurrently with the behaviour of states, with which these static reactions are associated. For example, a static reaction in the TAPERECORDER state can be expressed by adding a third concurrent part to it with a single state and a transition looping in this state. In order to represent static reactions in other states, such as RECORDING, those states have to be converted to AND-states. With this transformation, static reactions can be tested similarly to ordinary transitions.

3.4 Test data generation

Since the aim of the statechart testing method is to test a transition diagram, it is assumed that all labels are implemented correctly. Thus, an implementation may contain a different number of states, transitions traversing them in any way but labels of implemented transitions behave the same as those in a specification. Since sequences of labels generated by the test method are often not related to those taken by a system during routine operation, it is necessary to force those labels to be triggered through changes to externally accessible variables and all other labels — not to be triggered, even if they become such as a consequence of actions of some transitions. In addition, for every executed transition some output changes have to be observed, giving evidence that a transition with the expected label was actually taken by the implementation under test. The requirement of being able to trigger, the one of the input-output pair to identify a transition label, a requirement of absence of shared labels between states of a specification, and the one that transitions cannot directly enter default connectors, comprise the *design for test* condition. Under assumption that labels *rec*, *pause* and *continue* are triggered by the same *rec* event and an output from them makes it possible to uniquely identify them, the controller satisfies these requirements. More complicated statecharts may have to be designed in the first place to satisfy this condition; it is always possible to add extra inputs, outputs [20], slightly modify labels [6] and re-route transitions to default connectors [5] to satisfy design for test.

Reference [28] considers labelled transition systems; such a system deadlocks if no transition with a label attempted by a tester exists from its current state. The method presented in this paper converts labels to inputs; in response to an input triggering a label with no corresponding transition from its current state, a statechart would either take a different transition, a static reaction or simply ignore such an input. An output from an implementation (or absence of any) would then indicate which transition or static reaction (if any) has been executed.

If these requirements, those provided in Sect. 2.2, as well as the assumptions of the minimality (Sect. 3.1) of every OR-state in a specification, availability of a reliable reset (for both a specification and an implementation) and a known upper bound on a number of states in an implementation are satisfied, a test set can be generated, applied and provide a provable compliance of the behaviour of an implementation, to that of a specification [5]. It is important to stress the requirement of the synchronous behaviour of statecharts postulated in Sect. 2.2. Its purpose is to prevent uncontrollable sequences of transitions taken by a system under test. As a consequence, communication between concurrent states has to be absent. For some statecharts, it could be possible to test the core transition structure with transitions which do not trigger others and then test the remaining part of it, following the

approach described in [16].

4 Testing of statecharts using the Wp method

This section describes changes to the above testing method for statecharts, so as to use the Wp method as a foundation.

4.1 Description of the Wp method

The Wp method [12] is an improvement of the W one, targeted at the reduction of a number of test sequences. Instead of using a single W set, multiple smaller sets are introduced, each identifying a specific configuration. By ending test sequences with smaller sets, the number of test sequences is reduced. Unfortunately, in a faulty implementation small identification sets may fail to identify configurations correctly. To cope with this, a two-phase approach is taken where the first stage tests a part of a statechart and checks whether the small sets identify configurations; the rest of the statechart is tested at the second stage.

Formally, for a configuration *conf*, an *identification set* w^{root}_{conf} is a set allowing one to distinguish between *conf* and all other configurations in a statechart. The purpose of the *root* in the superscript of w will be explained later.

The first phase of the Wp method corresponds to the part of the W method where every configuration is entered and verified using the full $W = \cup_{conf} w^{root}_{conf}$ set. Consequently, each configuration *conf* is also checked whether it could be identified by the smaller set w^{root}_{conf}. The set of test cases used in the first phase can be written as

$$T_1 = C^M * (\{1\} \cup \Phi^M \cup (\Phi^M)^2 \cup \ldots \cup (\Phi^M)^{m-n}) * W$$

This differs from the full test set (Eqn. 1) in that the highest power of Φ^M is $m-n$ rather than $m-n+1$.

In addition to configuration verification, this phase also tests many transitions with labels used in C. Specifically, transitions labelled by singleton sequences in C get tested (between the respective configurations), since both their initial and final configurations are verified with the full W set. For non-singleton sequences, such as *rec pause*, the intermediate configuration does not necessarily get verified and thus in general none of the two transitions get tested during this phase. If, however, the *rec* label exists in C as a singleton sequence, the intermediate configuration is verified with W and thus both transitions get tested. This can be generalised, such that for a prefix-closed C (for every sequence $s \in C$, C contains all prefixes of s) all transitions traversed by C get tested between their respective configurations. This holds for the tape recorder.

At the second phase all transitions which were left out in the first phase are tested, using small sets w^{root}_{conf} to identify configurations and therefore create less test cases compared to the W method while still providing the same level of confidence in the result of testing. Let the initial configuration of a statechart be denoted by $conf_{init}$. The set of test cases for the second phase of the Wp method (without leaving out already tested transitions) is the following:

$$T_2 = \bigcup_{path \in TS} \{path\} * w^{root}_{CE(path, conf_{init})}$$

where $TS = C^M * \Phi^M * (\{1\} \cup \Phi^M \cup (\Phi^M)^2 \cup \ldots \cup (\Phi^M)^{m-n})$ and *CE* stands for Configuration Entered. $CE(path, conf)$ is the configuration entered after taking a path *path* from a configuration *conf*. For instance, with $conf_{init} = \{\text{STOP}, \text{IDLE}\}$, $w^{root}_{CE(rec\,pause, conf_{init})} = w^{root}_{\{PAUSE, IDLE\}}$ (only basic states in these two configurations are shown).

4.2 Merging rules for identification sets

In this subsection sets w_s^{root} for every state s of a statechart, merging rules for them and the construction of w_{conf}^{root} are described.

Let $w_{i,j}$ denote a set of paths which distinguishes between states i and j in the same flat statechart. For example, $w_{STOP,PLAY} = \{play\}$. For a state $s \in \rho(st)$, an *identification set* is defined as $w_s = \bigcup_{i \in \rho(st)} w_{s,i}$, which is a set allowing one to distinguish between a particular state s in a statechart st and all other states in st (but generally not those in a different state). For the controller these sets are: $w_{STOP} = \{stop\}, w_{PLAY} = \{direction\}, w_{RECORD} = \{play, rec\}$, $w_{REW\_FF} = \{rew\_or\_ff\}, w_{RECORDING} = \{pause\}, w_{PAUSE} = \{continue\}$. The characterisation set W can be expressed in terms of $w_{i,j}$ as

$$W_{st} = \bigcup_{s,i \in \rho(st)} w_{s,i} = \bigcup_{s \in \rho(st)} w_s \tag{2}$$

Rules to construct a characterisation set W^M for a statechart involve merging W_{MAIN} of the main statechart and merged W_s^M sets for every non-basic state s in it. The same approach can be applied to merging of identification sets. As defined above, a state st in some statechart can be identified with w_{st}; using S to denote this statechart, it is possible to write $w_{st}^S = w_{st}$. In order to obtain the identification set w_s^S for a state s contained in st of a higher-level state S (assuming the last two are OR states), one has to identify s in st and st in S. This gives $w_s^S = w_{st}^S \cup w_s^{st}$. Consequently, $w_s^S = w_{s_1}^S \cup w_{s_2}^{s_1} \cup \ldots \cup w_s^{s_n}$ where $s_1 \ldots s_n$ are such that $s_1 \in \rho(S), s_2 \in \rho(s_1), \ldots, s_n \in \rho(s_{n-1}), s \in \rho(s_n)$. For Fig. 1, $w_{PAUSE}^{MAIN} = w_{RECORD}^{MAIN} \cup w_{PAUSE}^{RECORD} = \{stop, play, continue\}$. The proposition in the following subsection describes how to reduce this set. The general rule for merging of w sets can be written as follows:

$$w_s^S = \begin{cases} \varnothing \text{ if } S = s \text{ or } S \text{ is an AND-state.} \\ w_s \text{ if } s \in \rho(S) \text{ and } S \text{ is an OR-state,} \\ \qquad \text{where } w_s \text{ identifies } s \text{ within its enclosing one } S. \\ w_{st}^S \cup w_s^{st} \text{ for some } st \in \rho^+(S) \text{ and } s \in \rho(st). \end{cases}$$

Above, $\rho^+(S)$ means a set of children, grandchildren and so on of S. Identification of a state s in the whole statechart is accomplished using w_s^{root}.

The identifying set for a configuration *conf* can be defined as a union of identifying sets of basic states in the configuration, i.e.

$$w_{conf}^{root} = \bigcup_{s \in conf,\ s \text{ is basic}} w_s^{root}$$

Applying this to the tape recorder, the size of the set of test cases for the first phase of the Wp method is 32 and the second one — 306, resulting in 338 sequences which is a half of the set provided in Sect. 3 above; the reduction could be much bigger for complex systems, where the number of transitions to verify is high.

4.3 Optimisation of state identification sets

Above, $w_{RECORDING}$ and w_{PAUSE} were merged with w_{RECORD} even though it was not necessary as $w_{RECORDING}$ and w_{PAUSE} already identify states in the flattened MAIN state. Note that if $w_{RECORDING} = \{continue\}$ was used, there would be no way to tell STOP and RECORDING apart in the merged statechart because transitions with the *continue* label exist from neither of them. These considerations give rise to the following proposition.

Proposition *Merging rules for state identification sets w_s within an OR state parent(s) can be simplified to $w_s^{root} = w_s^{parent(s)}$ if the following holds:*

- *Labels used in any state of a statechart are not used in any other one.*
- *For all states s of a statechart, w_s only contains labels which exist on transitions from s (described above).*

The first condition of this proposition implies, for instance, that *rec* cannot be used inside the RECORD state as it is used in the MAIN one. This condition holds due to the requirements for the statechart testing method described in Sect. 3.4; the second one can be made true by construction of w_s. Using the proposition,
$w^{MAIN}_{RECORDING} = \{pause\}, w^{MAIN}_{PAUSE} = \{continue\}$.

4.4 Optimisation of configuration identification

Consider the configuration with basic states PAUSE and IDLE. The identification set for it is $w_{\{PAUSE,IDLE\}} = \{continue, rew\_or\_ff\}$, but it could be reduced to a single element *continue-rew_or_ff* by taking the two transitions in the same step. The rule for construction of an identifying set for a configuration can thus be optimised as follows:

$$w^S_{conf} = \begin{cases} w^S_s \cup w^s_{conf} \text{ if } S \text{ is an OR-state, where } s \in \rho(S) \cap conf, \\ \quad \text{i.e. the only substate of } S \text{ in the configuration } conf. \\ \varnothing \text{ if } S \text{ is a basic state.} \\ w^{s_1}_{conf} \underline{\cup} w^{s_2}_{conf} \underline{\cup} \ldots \underline{\cup} w^{s_n}_{conf} \text{ if } S \text{ is an AND-state,} \\ \quad \text{for } n \text{ substates of } S, s_i \in \rho(S) \cap conf. \end{cases}$$

where $w^{s_a}_{conf} \underline{\cup} w^{s_b}_{conf} = \{seq_a \diamond seq_b \mid seq_a \in w^{s_a}_{conf} \cup \{1\}, seq_b \in w^{s_b}_{conf} \cup \{1\}\} \setminus \{1\}$, such that seq_a and seq_b can be taken in the same step. The purpose of w^S_{conf} is to identify a part of a configuration *conf* underneath S, thus it is assumed that $w^S_{conf} = w^S_{conf \cap \rho^+(S)}$. For example, when applied to the configuration *cfg* with basic states PAUSE and IDLE, the above definition becomes

$$w^{root}_{cfg} = w^{root}_{TAPERECORDER} \cup \underbrace{w^{TAPERECORDER}_{\{MAIN,RECORD,PAUSE\}}}_{= w^{MAIN}_{RECORD} \cup w^{RECORD}_{PAUSE}} \underline{\cup} \underbrace{w^{TAPERECORDER}_{\{SEARCH,IDLE\}}}_{= w^{SEARCH}_{IDLE}}$$

The aim of the rule is to take as many sequences in the same step as possible. For sets optimised as described in Sect. 4.3, all pairs of sequences from any two sets $w^{s_a}_{conf}$, $w^{s_b}_{conf}$ $(s_a, s_b \in \rho(S))$ can be taken in the same step, but applicability of configuration identification does not depend on whether state identification sets are optimised or not. To avoid redundant sequences in w^S_{conf} for an AND-state S, every element of the identification sets $w^{s_i}_{conf}$ has to be used only once (not shown in the definition). One of the seq_a and seq_b above is allowed to be empty (denoted *1*) to accommodate pairs of sequences which cannot be taken in the same step as well as to handle varying numbers of sequences in the sets to be $\underline{\cup}$-ed.

4.5 Merging rules for the characterisation set

The definition of W (Eqn. 2) for the case of merged w can be shown to be consistent with merging rules for W, such that after using the provided merging rules to construct w^{root}_s, it is possible to unite these sets afterwards (Eqn. 2) and obtain the rule for W construction (sections 3.2 and 3.3). The proof [5] is omitted from the paper. For example, W for the MAIN state of the tape recorder in Fig. 1 is $W = w^{MAIN}_{STOP} \cup w^{MAIN}_{PLAY} \cup w^{MAIN}_{RECORD} \cup w^{MAIN}_{RECORDING} \cup w^{MAIN}_{PAUSE} = \{stop, direction, rew\_or\_ff, pause, continue\}$. This W set and the reduced w_s were used in the computation of the size of the set of test cases in Sect. 4.2.

The described optimisation of configuration-identifying sets by taking multiple transitions at the same time may lead to a bigger W set than that constructed using the W method. For example, in the controller every configuration can be identified by a pair of transitions from its basic states. Optimised identification sets could use pairs of such transitions *continue-rew_or_ff, pause-rew_or_ff* and so on, a union of which is $\{stop, direction, pause, continue\} \underline{*} \{rew\_or\_ff, stop\_rew\_or\_ff\}$, containing 8 elements as opposed to 4 for W constructed from unoptimised identification sets. Such a growth of W causes the number of sequences in the first phase of the Wp method to increase to 64 but in effect actually leads to a reduction in the total size of the set of test cases. This follows from the w_{conf} sets being reduced twice and thus halves the size of the second phase. In the overall, the reduction is by 36% from the Wp method without optimisation of w_{conf} (but with optimisation of state identification) and by 68% from the W method.

4.6 Usage of status information

In some statecharts, it is possible to detect a state not by taking transitions but by observing values of some variables such as dashboard lights. This could lead to a considerable reduction of the size of a test set. Information obtained by observation is further called *status* information. Its usage is not new - most state-based object-oriented testing methods rely on it, such as [1, 30].

Status variables have to be correctly implemented and their usage restricted in the same way as labels: if some variable is used to identify a group of states, all those states should be within the same higher-level state. For example, information about whether a tape moves or not, cannot be used as a status as it is present in both PLAY and REW_FF states (which belong to different higher-level states) because one cannot independently identify states in MAIN or SEARCH with it.

Since status makes it possible to identify groups of states without taking any transitions, when constructing w_s for a state s, it is necessary to distinguish it only from members of the group it belongs to. More precisely, when identifying states, one could rely on $status_{gr}$ to identify groups gr of states and then construct $w_{s,gr}$ for states within each group. This leads to usage of a pair $(status_{gr}, w_{s,gr})$ instead of a set w_s. For example, in the tape recorder a tester can see whether a head is close to a tape; this is assumed to be unaffected by the state in the SEARCH statechart, which only controls the speed of tape movement. The head position permits an introduction of the *status_headclose* boolean status variable, considered to be true in the PLAY and RECORD states of the MAIN statechart and define $w_{STOP} = (\{\neg status\_headclose\}, \varnothing)$. $\neg$ means that the output should be negative from the status. The negation sign is not actually used in the set of test inputs but is shown here for clarification of the expected output. Note that the proposition in Sect. 4.3 is easily applicable to status variables. Here the length of test sequences for identification of the STOP state as well as the length of those in the W set of the first phase is reduced by one.

There could be more than a single status variable; for example, if one can observe the red 'recording in progress' light, the RECORD state can be assumed and $w^{MAIN}_{PLAY} = (\{status\_headclose, \neg\ status\_recordinglight\}, \varnothing)$. Here both STOP and PLAY states are identified without taking any transitions. $W_{MAIN} = (\{status\_headclose, status\_recordinglight\}, \{pause\})$ can be used for testing of the statechart. The speed of tape movement could also be used to tell REW_FF from the IDLE state. In the example, status information only reduces the size of the first phase of the Wp method, however in complex systems with configuration identification sets containing a number of labels, one could expect it to reduce the complexity of testing significantly.

Having constructed $status_{gr}$ and the corresponding $w_{s,gr}$ state identification set, it is necessary to integrate them. Since one can check all status variables in a single step, they

all can be united into a singleton sequence, the only element of it performing all status reporting. Variables and sequences of transitions used to identify states can be merged to identify configurations using the rules described in Sect. 4.2–4.4. Semantics of Harel statecharts makes it possible to observe status variables at the same time as applying the first element of a sequence from a state identification set. In practice, however, a tester is more likely to make a separate step devoted to observation of a status report. A configuration *conf* can be identified by a set $\{status_{conf}\}\underline{*}w^{root}_{status_{conf},conf}$ instead of a larger w^{root}_{conf}. Here $status_{conf}$ is the set of status variables, used in the identification of this configuration; if a particular value of $status_{conf}$ does not single out *conf*, $w^{root}_{status_{conf},conf}$ is used to identify it among configurations satisfying that value of $status_{conf}$. The $\underline{*}$ operator is used instead of $*$ in order to observe status and take a transition labelled by the first element of sequences in $w^{root}_{status_{conf},conf}$ at the same time.

Certain aspects of consistency of statecharts can be verified by testing. For example, it is possible to ascertain that if a state is entered, its parent state is entered too. For the tape recorder, one could verify RECORDING $\in \rho$(RECORD) by using $w_{RECORDING} = (\{status\_recordlight\}, \{pause\})$, since $w_{RECORD} = (\{status\_recordlight\}, \varnothing)$ and from the proposition in Sect. 4.3 $w_{RECORDING} = (\varnothing, \{pause\})$. Status information provides a 'free' way to do this type of checking.

5 Conclusion

The Wp method can be applied to statecharts in a similar way to the W one [6, 5, 8], by extension of merging rules for identification of a configuration. Due to the complexity of statecharts, the Wp method can be expected to provide a significant reduction in the size of the set of test cases, compared to the W method. Further, rules were provided for different optimisations of such sets, allowing additional reduction in the complexity of testing. Direct observation of state information can lead to further reduction of the size of a set of test cases and can be applied together with the other optimisations described. Usage of status variables also permits verification of state properties of a statechart, such as whether a parent of an entered state is itself in a configuration.

The most important limitations of the presented extension are lack of handling of interlevel transitions, requirement for labels not to be shared between states and a need to force sequences of transitions during test execution. For the first of these, one might consider interlevel transitions to belong to their scope states; shared labels can be overcome either by partial flattening of the structure of a statechart or by making TCB construction sensitive to shared labels. The work on these two issues is close to completion. For the last problem, constraint logic programming [23] could potentially be used to derive sequences of test inputs without resorting to artificial test inputs. This is a possible direction for future work.

Theoretical foundations of the usage of W and Wp testing methods for statecharts have been shown to be correct in [5].

Acknowledgements

This work was funded by the DaimlerChrysler Research Laboratory (FT3/SM), Berlin, Germany.

References

[1] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. White. State generation and automated class testing. *Software Testing, Verification and Reliability*, 10(3):149–170, September 2000.

[2] M. Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, and Y. Wolfsthal. A feasibility study in formal coverage driven test generation. In *36th Design Automation Conference (DAC'99)*, June 1999.

[3] M. Blackburn. Using models for test generation and analysis. In *Digital Avionics Systems Conference (DASC'98)*, 1998.

[4] M. Blackburn, R. Busser, and J. Fontaine. Automatic generation of test vectors for SCR-style specifications. In *12th Annual IEEE Conference on COMPuter ASSurance, COMPASS '97*, 18–19 June 1997.

[5] K. Bogdanov. *Automated testing of Harel's statecharts*. PhD thesis, The University of Sheffield, January 2000.

[6] K. Bogdanov and M. Holcombe. Statechart testing method for aircraft control systems. *Software testing, verification and reliability*, 11:39–54, 2001.

[7] K. Bogdanov and M. Holcombe. Properties of concurrently taken transitions of statecharts. In *Semantic Foundations of Engineering Design Languages (SFEDL)*, 2002.

[8] K. Bogdanov, M. Holcombe, and H. Singh. Automated test set generation for statecharts. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods - FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 107–121. Springer Verlag, 1999.

[9] T. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

[10] 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*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Formal Methods Europe, Springer Verlag, April 1993.

[11] J.-C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. *Lecture Notes in Computer Science*, 1102:348–359, 1996.

[12] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591 – 603, June 1991.

[13] D. Harel, H. Lachover, A. Nammad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[14] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[15] R. M. Hierons. Testing from a Z specification. *Journal of software testing, verification and reliability*, 7(1):19–33, 1997.

[16] R. M. Hierons. Testing from semi-independent communicating finite state machines with a slow environment. *IEE Proceedings on Software Engineering*, 144(5–6):291–295, 1997.

[17] M. Holcombe and F. Ipate. *Correct Systems: building a business process solution*. Springer-Verlag Berlin and Heidelberg GmbH & Co. KG, September 1998.

[18] H. Hong, Y. Kim, S. Cha, D. Bae, and H. Ural. A test sequence selection method for statecharts. *Software testing, verification and reliability*, 10:203–227, 2000.

[19] F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. *International Journal on Computer Mathematics*, 63:159–178, 1997.

[20] F. E. Ipate. *Theory of X-machines and Applications in Specification and Testing*. PhD thesis, University of Sheffield, July 1995.

[21] A. Kerbrat. Automated test generation from SDL/UML specifications. Obtained privately, March 1999.

[22] Y. Kim, H. Hong, S. Cho, D. Bae, and S. Cha. Test cases generation from UML state diagrams. *IEE Proceedings - Software*, 146(4):187–192, August 1999.

[23] H. Lötzbeyer and A. Pretschner. Testing concurrent reactive systems with constraint logic programming. In *2nd Workshop on Rule-based constraint reasoning and programming, Singapore*, September 2000.

[24] E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel. On formal semantics of statecharts as supported by Statemate. In *BCS-FACS Nothern Formal Methods Workshop*, pages 0–13, Craiglands Hotel, Ilkley, West Yorkshire, U.K., July 1997.

[25] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *Second International Conference on the Unified Modeling Language (UML99), Fort Collins, CO*, October 1999.

[26] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. `http://www.informatik.uni-bremen.de:80/~jp/papers/sacj97.ps.gz`, August 1996.

[27] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. Meyer, editors, *Int. Conf. TACS'91: Theoretical aspects of Computer Software*, volume 526, pages 244–264. Springer-Verlag, September 1991.

[28] Q. M. Tan, A. Petrenko, and G. von Bochmann. Checking experiments with labeled transition systems for trace equivalence. In *IFIP 10th International Workshop on Testing of Commnication Systems (IWTCS'97)*, Korea, 1997.

[29] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[30] B. Tsai, S. Stobart, N. Parrington, and I. Mitchell. Automated class testing: Using threaded multi-way trees to represent the behavior of state machines. *Annals of Software Engineering*, 8:203–221, 1999.

Testing Nondeterministic (stream) X-machines

Florentin Ipate
Faculty of Sciences, Pitesti University
Str Targu din Vale 1, 0300 Pitesti, Romania

Marian Gheorghe
Department of Computer Science, Sheffield University
Regent Court, Portobello Street, Sheffield, S1 4DP, UK

Mike Holcombe
Department of Computer Science, Sheffield University
Regent Court, Portobello Street, Sheffield, S1 4DP, UK

Tudor Bălănescu
Faculty of Sciences, Pitesti University
Str Targu din Vale 1, 0300 Pitesti, Romania

Abstract. Introduced by Eilenberg, *X-machines* (or *Eilenberg machines*) were proposed by Holcombe in 1988 as a basis for a possible specification language and since then a number of further investigations have demonstrated the value of this idea. A number of classes of X-machines have been identified and studied, most importantly the class of *stream X-machines*. A theory of testing based on stream X-machines has also been developed. This allows the generation of test sets that are proved to guarantee the correctness of implementation against the specification under certain circumstances. A recent paper generalises this theory to the general X-machine model and, furthermore, the results proven can form the basis of two distinct testing strategies. This paper generalises these recent results to the non-deterministic case. The generalisation is non-trivial and it can cope with all types of non-determinism that can be found in an X-machine model.

Keywords: Testing, finite state machine, (stream) X-machine, formal specification
CR Categories: D.2.2, D.2.4

1 Introduction

One of the strengths of developing a formal specification of a system is the fact that it can act as a reference point for the project development, it can define, in the form of a quasi-legal statement, the required outcome of the project and it can also be a source of information that can be used to establish that the implementation is correct. In recent years there has been some interest in trying to use the information in a formal specification as a basis for test set generation. Testing is all about finding faults and very seldom the issue of the number of faults that remain in the implementation after testing is discussed. Bernot et al. [6] and Dauchy et al. [11] consider the generation of test sets from algebraic specifications, here there is a more comprehensive framework (the "hypothesis") which allows for the issue of test effectiveness to be discussed but the test generation process does not exploit this particularly.

The problem of test effectiveness is best addressed if the test set can be guaranteed to find *all faults* of the implementation. One approach is to consider two algebraic objects (the specification and the implementation), each of them characterised by an input/output behaviour, and to *prove* that, if the behaviours of these objects coincide for any input in the test set, they will coincide for any input in the domain. Thus, the specification and the implementation will be guaranteed to have identical behaviour provided that they behave identically when supplied with the inputs in the test set. Obviously, such an approach would only be applicable to certain systems and specification

languages; it will not be applicable to an arbitrary computer system (an arbitrary Turing machines), otherwise the halting problem for Turing machines, for example, will be contradicted.

This approach has been employed in the area of test generation for the software modelled by *finite state machines (FSM)* [10], [30], [14], [21]. The best known testing methods based on FSM specifications are transition-tour (T-) method [31], [34], unique-input-output (UIO-) method [33], [34], distinguishing sequence (DS-) method [15], [34] and characterizing set (W-) method [10], [34]. The purpose of any testing method is to come up with at least two requirements: (a) the test suite should be relatively short and (b) the test suite should cover as much as possible all faults [14]. Concerning fault coverage, the last two methods cover all faults. Shorter test suite has been obtained for UIO-method in the case of UIO sequences with overlapping [35], or either when the FSM has reset capacity (there is an input that takes every state to the initial state) or has loops (transitions with the same initial and final states) [1], or when the FSM contains either invertible transitions [16] or invertible sequences of transitions [17]. A variant of W-method called 'partial W-method' (or Wp-method) yields shorter test suites than the W-method [14]. All these methods assume that the control aspects of the software are separated from the system data and can be modelled by a FSM. However, in some situations it is very difficult to separate the system control from its data [10], so a more complex specification model that integrates these two aspects is needed.

Such a model is the *X-machine* (or *Eilenberg machine*), a blend of FSMs, data structures and processing relations (or functions). In its essence an X-machine is like a FSM but with one important difference. A basic data set, X, is identified together with a set of basic processing relations (or functions), Φ, which operate on X. Each arrow in the finite state machine diagram is then labelled by a relation (or function) from Φ, the sequences of state transitions in the machine determine the processing of the data set and thus the relation or function computed. The data set X can contain information about the internal memory of a system as well as different sorts of output behaviour so it is possible to model very general systems in a transparent way. Introduced by Eilenberg [12] in 1974, X-machines are proposed by Holcombe [19] as a basis for a possible specification language and since then a number of further investigations have demonstrated that the model is intuitive and easy to use [21], [24], [25], [13].

A number of classes of X-machines have been identified and studied [21], [23]. Typically, these classes are defined by restrictions on the underlying data set X and the set of basic processing relations (or functions), Φ, of the machines. Among all these, the class of *stream X-machines* (SXM) has received the most attention. Particular types of SXMs, having stacks [22], or sequences of symbols [3] as memory, or output sequences instead of single output symbols [2], have also been considered.

Thus SXMs are generalisations of finite state machines, similar to extended finite state machines (EFSM) [29], [9]: here, the variables are replaced by a memory and the sets of predicates and assignments are replaced by a set of processing relations (or functions).

A testing strategy for systems specified by stream X-machines has also been developed [23], [20], [21]. Here, a number of transition are grouped into processing relations (or functions) and these are assumed to be implemented correctly. Thus, testing the SXM reduces to testing its transition diagram (the associated *automaton* of the stream X-machine). The correctness of the processing relations (or functions) is checked by a separate process: depending on the nature of the relation (or function), these can be tested using the same method or alternative functional methods [21], [24]. Furthermore, the method can only be applied if the processing relations (or functions) meet some "design for test conditions", completeness and output-distinguishability. The method was first developed in the context of deterministic stream X-machines [23] and then extended to the non-deterministic case considering test for equivalence [26] or conformance [18].

A recent paper [27] generalises the existing stream X-machine testing theory in more than one way. Firstly, it considers the general X-machine model and secondly, the results proved can give rise to two distinct testing strategies. These results and their corresponding testing strategies are then particularised to the stream X-machine class. One of these strategies is a slightly stronger form of the result given in [23].

However, this generalization is only aimed at *deterministic* X-machines. On the other hand, there is a practical need for testing *non-deterministic* models. Non-determinism and concurrency

are two important features of formal specification languages for communicating software, in particular communication protocols. All major specification languages for communication software - e.g. LOTOS [8], SDL [5] - support non-determinism. Moreover, a system of communicating deterministic X-machines may have non-deterministic behaviour [4].

This paper generalises these recent results to the non-deterministic case. The generalisation is non-trivial and it can cope with all types of non-determinism that can be found in an X-machine model. The paper is structured as follows: section 2 contains concepts from FSM theory and FSM testing; section 3 introduces the (non-deterministic) X-machine model and related concepts; sections 4 and 5 present the two testing strategies and the corresponding theoretical concepts and results; these are then particularised to the stream X-machine class in section 6, finally conclusions are drawn in section 7.

2 Finite state machine concepts

This section defines the *automaton* or *recognizer* and the *finite state machine* [28] and presents related concepts and results that will be used later in the paper.

Before we go any further, we introduce the notation used in this paper. When considering sequences of inputs or outputs we will use A^* to denote the set of finite sequences with members in A. λ will denote the empty sequence and $A^+ = A^* - \{\lambda\}$. For $a, b \in A^*$, ab will denote the concatenation of sequences a and b. For $U, V \subseteq A^*$, $UV = \{ab \mid a \in U, b \in V\}$. For a non-empty sequence $a = a_1 \dots a_n$ with $a_1, \dots, a_n \in A^*, n \geq 1$, $rear(a) = a_n$ denotes the rightmost element of the sequence; $rear(\lambda)$ is undefined.

For a relation (or a partial function) $f : A \longleftrightarrow B$, dom(f) denotes the domain of f. For $U \subseteq dom(f)$, $f(U) = \{b \mid \exists a \in U \text{ with } afb\}$. For two (partial) functions $f : A \longrightarrow B$, $g : B \longrightarrow C$, $g \circ f : A \longrightarrow C$ is a function defined by $g \circ f(a) = g(f(a))$ $\forall a \in A$.

For a finite set A, $card(A)$ denotes the number of elements of A.

Definition 1. *An* automaton *is a system $A = (\Sigma, Q, F, q_0, T)$ where:*
Σ, Q, T are the finite set of inputs, states *and* final states, *$T \subseteq Q$; F is the* next state *function, $F : Q \times \Sigma \longrightarrow 2^Q$, and $q_0 \in Q$ is the* initial state.

For the testing purposes all the states are final states, $T = Q$, and thus the set T will be ignored.

Note 1. In general, an automaton may be *non-deterministic* in the sense that for each state q and input σ there may be more than one next state. An automaton is called *deterministic* if F is a (partial) function $F : Q \times \Sigma \longrightarrow Q$.

Definition 2. *If $q, q' \in Q$, $\sigma \in \Sigma$ and $q' \in F(q, \sigma)$ we say that σ is an* arc *from q to q' and write $\sigma : q \to q'$. If $q, q' \in Q$ are such that there exist $q_1, \dots, q_{n+1} \in Q$, $n \geq 0$, with $q_1 = q$, $q_{n+1} = q'$ and $\sigma_i : q_i \to q_{i+1}$ for all $1 \leq i \leq n$, we say that we have a* path *$s = \sigma_1 \dots \sigma_n$ from q to q' and write $s : q \to q'$. For $q \in Q$, $L_q = \{s \in \Sigma^* \mid \exists r \in Q \text{ such that } s : q \to r\}$ is called the* language accepted by A in q. *If $q = q_0$ then L_q is simply called the language* accepted by A *and is denoted by L_A.*

Two automata A and A' are called $U-$equivalent, $U \subseteq \Sigma^*$ if $L_A \cap U = L_{A'} \cap U$. If $U = \Sigma^*$ then A and A' are called equivalent.

Definition 3. *A* finite state machine (FSM) *is a system $M = (\Sigma, O, Q, F, \lambda)$ where:*
Σ, O, Q are finite, non-empty sets of inputs, outputs *and* states, *respectively; F is the* next state *function as in Definition 1 and $\lambda : Q \times \Sigma \longrightarrow O$, is the* output function.

Definition 4. *Given a set of input sequences $U \subseteq \Sigma^*$, two states p, q are called $U-$equivalent if p and q respond with identical output sequences to each input sequence in U.*

Two FSMs M and M' are called $U-$equivalent, $U \subseteq \Sigma^*$ if their initial states are $U-$equivalent. Two FSMs are equivalent if they are $U-$equivalent for any set U of input sequences.

We now turn our attention to FSM testing and in particular to the generation of test sequences from a FSM specification. Let us consider the automaton associated with an X-machine (see Note 3). This automaton is deterministic, but in general not completely specified. The testing theory based on FSMs, that interests us, deals in general with FSMs deterministic, minimal, completely specified and refers to an equivalence relationship between specification and implementation (see [10], [14] and [32] for a review on FSM testing). According to this theory, once we have a specification S modelled as a FSM with the above properties we may construct a test set U and when applied to the implementation I is able to find all errors if the implementation I is assumed to be modelled as a FSM [10], [14]. Formally if both S and I are modelled as FSMs and satisfy some conditions, such as those previously mentioned, then they are equivalent iff they are $U-$equivalent, for U a test set as above. When dealing with an X-machine, an automaton $A(P)$ (see Note 3) may be always defined. For two X-machines P and P', we are interested in finding a set U over Φ^*, such that whenever their underlying automata $A(P)$ and $A(P')$ are $U-$equivalent, see Notes 3, then they equivalent. There have been suggested some ways of getting a FSM from an automaton [7]. According to [7] an incompletely specified automaton is transformed into a completely specified FSM by considering for every transition of the automaton an output 1; for all new transitions introduced for missing inputs and going to a dead state, an output 0 is associated with. For the obtained FSM, which must be deterministic and minimal, a test set U may be built according to a given testing strategy [10], [14]. Further on, U may be used to prove the equivalence of the two automata representing the underlying state machines of the specification and implementation X-machines.

3 X-machine concepts

This section presents the X-machine model and illustrates it with an example.

Definition 5. *An* X-machine *is a system* $P = (X, Q, \Phi, F, q_0, c, d, O, o)$ *where:*

- X *is a (possibly infinite) set called the* data set;
- Q *is the finite* set of states;
- Φ *is the set of* basic processing relations, *a finite set of partial functions* $f : X \longleftrightarrow X$
- F *is the* next state *function, a (partial) function* $F : Q \times \Phi \longrightarrow 2^Q$;
- $q_0 \in Q$ *is the* initial state*;*
- c *is the* stopping condition *of* P, *a predicate on* X;
- d *is the* testing domain *of* P, *a function* $d : \Phi \longrightarrow 2^X$ *such that* $\forall f \in \Phi \; d(f) \subseteq dom(f)$;
- O *is the* output set;
- o *is the* output function *of* P, *a (partial) function* $o : X \longrightarrow O$ *with* $dom(o) \supseteq dom(\Phi)$, *where* $dom(\Phi) = \bigcup_{f \in \Phi} dom(f)$

Compared with the previous definition of an X-machine [21], the above definition includes some specific elements introduced for testing purposes: stopping condition (c), testing domain (d), output function (o); d determines the doamain values of each processing function that can be used in testing, o the output that can be observed, whereas c replaces the set of final states that usually appears in previous definitions of an X-machine.

Note 2. For an X-machine P as above, o is extended to a free-semigroup morphism $o : X^* \longrightarrow O^*$. Thus $o(\lambda) = \lambda$, and $o(x_1 \ldots x_p) = o(x_1) \ldots o(x_p), p \geq 1$. In what follows we will refer to this extension.

It is sometimes helpful to think of an X-machine as an automaton with the arcs labelled by symbols from Φ.

Note 3. For an X-machine P as above, the *associated automaton* is $A(P) = (\Phi, Q, F, q_0)$.

Example 1. Let us imagine a computer program that, when supplied with a string containing any letter, digit and the blank character, counts the number of words for which the first character is 1, all prefixes have a number of 1s that is greater than or equal to the number of 0s and characters others than 0 or 1, may count as either a 0 or an 1. Words are separated by one or more blanks. For example, the input sequence $'10a10\$\$01b01\$111'$ where \$ denotes a blank, can be split into the following words $'10a10'$, $'01b01'$ and $'111'$. If the character a in the first word counts as an $'1'$ then the program will count two such words, otherwise, it will count only one word. An X-machine model of this program is given next. Q is $\{q_0, q_1, q_2, q_3\}$ and F is defined by $F(q_0, i) = q_3$, $F(q_3, e) = q_2$, $F(q_3, f_1) = q_1$, $F(q_3, f_b) = q_3$, $F(q_1, f_0) = q_2$, $F(q_1, f_b) = q_3$, $F(q_1, g) = q_1$, $F(q_2, f_b) = q_3$, $F(q_2, h) = q_2$. $X = \Sigma^* \times N' \times N$, where $\Sigma = \{0, 1, b\}$, $N' = N \cup \{-1, -2, -3, -4\}$. The data set of the machine comprises tuples $(s, n, k), s \in \Sigma^*$, $n \in N'$, $k \in N$, where s is the string supplied to the system, n represents the difference between the number of 1s and the number of 0s in the current word and k the number of words. For reasons that will be explained later, n may also have the "dummy" values -1, -2, -3 or -4. $\Phi = \{i, e, f_0, f_1, f_b, g, h\}$, where the definitions of the processing relations are as follows:

$(s, -4, 0)i(s, 0, 0)$, where $s \in \Sigma^*$,
$(ys, 0, k)e(s, -1, k)$, where $s \in \Sigma^*$, $k \in N$, y is any character other than 1,
$(0s, 0, k)f_0(s, -2, k-1)$, where $s \in \Sigma^*$, $k \in N, k > 0$,
$(1s, 0, k)f_1(s, 1, k+1)$, where $s \in \Sigma^*$, $k, n \in N$,
$(\$s, n, k)f_b(s, 0, k)$, where $s \in \Sigma^*$, $k \in N, n \in N' \setminus \{-4\}$,
$(xs, n, k)g(s, n+y, k)$, where $s \in \Sigma^*$, $k, n \in N$, $(n > 0$, $x \in \{0, c\}$ and $y = -1)$ or $(x \in \{1, c\}$ and $y = 1)$, where c denotes any character other than 0 or 1,
$(xs, n, k)h(s, -3, k)$, where $s \in \Sigma^*$, $k \in N, n \in N' \setminus \{-4\}$, $x \in \Sigma$.

We take $O = N' \times N$, $o(s, n, k) = (n, k)$, $s \in \Sigma^*$, $n, \in N'$, $k \in N$; $c(\lambda, n, k) = true$ for all $n \in N', k \in N$, otherwise false.

It may be observed that g is a relation associating to any element (cs, n, k), where c is any character other than 0 or 1, either $(s, n-1, k)$ or $(s, n+1, k)$ and $(\$s, n, k) \in dom(h) \cap dom(f_b)$. Both of the above conditions are different forms of non-determinism as we will show further on.

Definition 6. *If $q, q' \in Q$, $f \in \Phi$, and $q' \in F(q, f)$ we say that f is an* arc *from q to q' and write $f : q \to q'$. If $q, q' \in Q$ are such that there exist $q_1, \ldots, q_{n+1} \in Q$, $n \geq 0$, with $q_1 = q$, $q_{n+1} = q'$ and $f_i : q_i \to q_{i+1}$ for all $1 \leq i \leq n$, we say that we have a* path *$pt = f_1 \ldots f_n$ from q to q' and write $pt : q \to q'$. Each path $pt = f_1 \ldots f_n$ gives rise to a relation (the* path relation*) $f_{pt} : X \longleftrightarrow X$ where $x f_{pt} x'$ if and only if $\exists x_1, \ldots, x_{n+1} \in X$ such that $x_i f_i x_{i+1}$ for all $1 \leq i \leq n$, where $x_1 = x, x_{n+1} = x'$.*

A machine computation takes the form of a traversal of a path in the state space and the application, in turn, of the path labels (which represent basic processing relations). This gives rise to the relation computed by the machine, as defined next.

Definition 7. *The relation computed by P, $f^P : X \longleftrightarrow X$ is defined by $x f^P x'$ if and only if $\exists q \in Q$ and a path $pt : q_0 \to q$ such that $x f_{pt} x'$ and $c(x')$ is true.*

In general, an X-machine may be non-deterministic, in the sense that the application of an initial data value x may produce more than one value of the data set. An X-machine is deterministic if its associated automaton is deterministic, Φ is a set of partial functions rather than relations, any two processing functions that emerge from the same state have disjointed domains and no processing function can process values for which the stopping condition is true.

Definition 8. *An X-machine P is called* deterministic *when $F : Q \times \Phi \longrightarrow Q$; Φ is a set of partial functions rather than relations; for any two distinct processing functions $f, g \in \Phi$, if $\exists q \in Q$ such that $(q, f) \in dom(F)$ and $(q, g) \in dom(F)$ then $dom(f) \cap dom(g) = \emptyset$; for any processing function f, if $x \in dom(f)$ then $c(x) = false$.*

According to the previous definition an X-machine can have 4 types of non-determinism:

– *state non-determinism* if there exist $q \in Q, f \in \Phi$ with $card(F(q, f)) > 1$.

- *operator non-determinism* if some elements of Φ are relations instead of partial functions.
- *domain non-determinism* if there exist $q \in Q, f_1, f_2 \in \Phi$ with $(q, f_1), (q, f_2) \in dom(F), f_1 \neq f_2$ and $dom(f_1) \cap dom(f_2) \neq \emptyset$.
- *termination non-determinism* if there exist $x \in X, f \in \Phi$ with $x \in dom(f)$ and $c(x) = true$.

The X-machine in Example 1 has domain and operator non-determinism but not state or termination non-determinism (i.e. the fact that c is *true* only for (λ, n, k) values avoids termination non-determinism).

In general state non-determinism and termination non-determinism are not really necessary since they can be removed by rewriting the X-machine. Indeed, state non-determinism can be removed by using standard algorithms that take a non-deterministic automaton and produce an equivalent deterministic automaton. The transformation does not preserve the domain determinism, as can be seen in Example 2.

Example 2. Let us consider an X-machine having the following next state function: $F(0, f) = \{1, 2\}, F(1, f_1) = 3, F(2, f_2) = 3$ where $dom(f_1) \cap dom(f_2) \neq \emptyset$. It is state non-deterministic but domain deterministic. An equivalent deterministic automaton is $F'(0, f) = 12, F'(12, f_1) = F'(12, f_2) = 3$. The new equivalent X-machine is state deterministic but domain non-deterministic.

Lemma 1. *For any X-machine P an equivalent termination deterministic X-machine P' may be constructed.*

Proof. If P is termination non-deterministic then P' will have the data set $X' = X \times \{0, 1\}$, where X is data set of P, and the same set of states with P. All the processing relations of P' will be defined on subsets of $X \times \{0\}$: $\Phi' = \{f' \mid \exists f \in \Phi, \forall x, y \in X$ if xfy then $(x, 0)f'(y, 0)\}$; $c'(x)$ will only be true iff $x \in X \times \{1\}$. For each state $q \in Q$, new loop-back transition will be added. All these transitions will be labelled by a relation t, that translates every $(x, 0) \in X \times \{0\}$ into $(x, 1) \in X \times \{1\}$. Consequently, $F' : Q \times \Phi' \longrightarrow Q$ is defined by $r \in F'(q, f')$ iff $r \in F(q, f)$ and $F'(q, t) = \{q\}$ for all $q \in Q$. The testing domain d' is defined by $d'(f') = \{(x, 0) \mid x \in d(f)\}$ and $d(t) = \{(x_t, 0)\}$ for a particular $x_t \in X$. Finally $o'(x, 0) = o(x)$, for all $x \in dom(o)$.

According to this lemma termination non-determinism can be transformed into domain non-determinism.

From this note it follows that we may easily transform any X-machine into a new one having associated stopping conditions to every state. This observations shows that the above transformation is on the one hand consistent with the testing hypothesis asking for all states being final states and on the other hand proves that stopping conditions may be ignored. Subsequently we will consider X-machines which are state and termination deterministic and without stopping conditions specified. According to this observation since now on any X-machine as introduced by Definition 5 will have the next state function of the form $F : Q \times \Phi \longrightarrow Q$ no stopping condition at all.

Domain non-determinism and operator non-determinism cannot be removed by rewriting the X-machine, so there seems no good reason to outlaw these types of non-determinism and, as illustrated in Example 1, they can be useful.

4 The breakpoint test set of X-machines

We turn now our attention to testing. As stated in the introduction, the approach used is to consider two X-machines and to generate a *finite* set of sequences that, when applied to the two machines with identical observable results, guarantees that the two machines have identical behaviour. This is formalised next. The section also introduces the concepts that are required in the testing process and shows how a test set can be generated.

Definition 9. *For an X-machine P, a partial function $h_P : X^* \longrightarrow X^*$ called a* a breakpoint computation of P *is defined as $h_P(x_1 \dots x_n) = y_1 \dots y_n$, if there exists a path $f_1 \dots f_n$ from q_0 such that $x_i f_i y_i$, $x_i \in d(f_i)$, $1 \leq i \leq n$, $n \geq 0$.*

Note 4. The elements x_i for all $2 \leq i \leq n$ in the above definition for which $x_i \neq y_{i-1}$ are called *breakpoint values* of h_P.

That is, a breakpoint computation associates a sequence of data values with the sequence of data values produced when a path in the machine is traversed. Not that, in general, for a non-deterministic X-machine, many paths may be traversed by one sequence of data values and furthermore a path may produce many sequences of data values, so a breakpoint computation may not be uniquely defined, as illustrated by the following example.

Example 3. For P as in Example 1 and

$x_1 = (11a0\$1, -4, 0)$, $x_2 = (11a0\$1, 0, 0)$, $x_3 = (1a0\$1, 1, 1)$, $x_4 = (a0\$1, 2, 1)$, $x_5 = (0\$1, 0, 1)$, $x_6 = (\$1, -2, 0)$, $x_7 = (1, 0, 0)$.

we may get either $y = y_1y_2y_3y_4y_5y_6y_7$, or $y' = y_1y_2y_3y_4'y_5y_6y_7$, if path $if_1ggf_0f_bf_1$ is used. Consequently we may define $h_P(x) = y$ or $h'_P(x) = y'$, where $x = x_1 \dots x_7$. The values y_i, $1 \leq i \leq 7$ are obtained as follows:

$y_1 = (11a0\$1, 0, 0)$ $(x_1iy_1$ and $F(q_0, i) = q_3)$,
$y_2 = (1a0\$1, 1, 1)$ $(x_2f_1y_2$ and $F(q_3, f_1) = q_1)$,
$y_3 = (a0\$1, 2, 1)$ $(x_3gy_3$ and $F(q_1, g) = q_1)$,
$y_4 = (0\$1, 3, 1)$, $y_4' = (0b1, 1, 1)$, $(x_4gy_4$, x_4gy_4' and $F(q_1, g) = q_1)$,
$y_5 = (\$1, -2, 0)$, $(x_5f_0y_5$ and $F(q_1, f_0) = q_2)$,
$y_6 = (1, 0, 0)$, $(x_6f_by_6$ and $F(q_2, f_b) = q_3)$,
$y_7 = (\lambda, 1, 1)$, $(x_7f_1y_7$ and $F(q_3, f_1) = q_1)$.

It may be observed that x_5 is a breakpoint value of both h_P and h'_P $(x_5 \neq y_4$, $x_5 \neq y_4')$.

In order to test non-deterministic implementations, one usually makes a so-called *complete-testing assumption*, [30] which says that it is possible, by applying a given sequence of data values s to a given implementation a finite number of times, to exercise all the paths of the implementation that can be traversed by s. Without such an assumption, no test suites can guarantee full fault coverage for non-deterministic implementations. In terms of X-machines, this means that the application of a set of sequences I to the implementation a finite number of times will result in a (breakpoint) computation of the implementation.

A set of sequences of data set values I will detect all the faults of an implementation P' against a specification P if the application of I a sufficient number of times (i.e. to exercise all the paths of P and P' that can be traversed by I) to P and P' with identical output results guarantees that P and P' will behave identically for any sequence of data set values. Such a set will be called a *(breakpoint) test set of P and P'.*

Definition 10. *Two X-machines P and P' are called* testing-compatible *if their data sets, sets of basic processing relations, and output functions coincide.*

Definition 11. *Let $P = (X, Q, \Phi, F, q_0, d, O, o)$ and $P' = (X, Q', \Phi, F', q_0', d, O, o)$ be two X-machines that are testing-compatible. Then a finite set $I \subseteq X^*$ is called a* breakpoint test set *of P and P' if whenever there exist h_P and $h_{P'}$ two breakpoint computations of P and P', respectively, with $o \circ h_P(x) = o \circ h_{P'}(x)$, $x \in I$ we have $f^P = f^{P'}$.*

Note 5. For two X-machines P and P', if $A(P)$ and $A(P')$ are equivalent automata then $f^P = f^{P'}$. However, the converse implication is not true, as it is easy to construct two testing compatible X-machines that compute identical relations and have non-equivalent associated automata.

Similar to the deterministic case [23], [27], the idea of our method is to prove a stronger requirement, that is that the two associated automata are equivalent. In this way, in order to test the two X-machines we can use the test sets of the associated automata. However, this idea can only work if it is possible to distinguish between any two processing relations using appropriate values.

Definition 12. *An X-machine is called* weak output-distinguishable *if for any $f, g \in \Phi$, if there exist $x, x_1', x_2' \in X$ with xfx_1', xgx_2' and $o(x_1') = o(x_2')$ then $f = g$.*

An X-machine is called strong output-distinguishable *if for any $f, g \in \Phi$, if there exist $x, x_1', x_2' \in X$ with xfx_1', xgx_2' and $o(x_1') = o(x_2')$ then $f = g$ and $x_1' = x_2'$.*

The X-machine in Example 1 is strong output-distinguishable; in order to meet this property the dummy values -1, -2, -3 and -4 were used in the definitions of e, f_0, h and i, respectively.

Obviously, if Φ is a set of (partial) functions rather than relations, weak output-distinguisability and strong output-distinguishability coincide. In this section only the "weak" condition will be used, the "strong" condition will be required in the next section.

We now need a mechanism for translating sequences of processing relations into sequences of data. This is a test function, as defined next.

Definition 13. *Let $P = (X, Q, \Phi, F, q_0, d, o)$ be an X-machine. Then a function $t^P : \Phi^* \longrightarrow X^*$ recursively defined as follows:*

- $t^P(\lambda) = \lambda$;
- *for $p \geq 0$, consider t^P defined for any string $f_1 \dots f_p \in \Phi^*$; then for any string $f_1 \dots f_p f_{p+1}$, $t^P(f_1 \dots f_p f_{p+1})$ is either of:*
  - *if $f_1 \dots f_p$ is a path in P that emerges from the initial state q_0 then $t^P(f_1 \dots f_p f_{p+1}) = t^P(f_1 \dots f_p)x_{p+1}$, where $x_{p+1} \in d(f_{p+1})$*
  - *otherwise $t^P(f_1 \dots f_p f_{p+1}) = t^P(f_1 \dots f_p)$.*

is called a test function *of P.*

In other words, for any sequence of processing relations $pt = f_1 \dots f_p$, $t^P(pt)$ is a sequence of data that exercises the longest prefix of pt that is a path of the machine and also tries to exercise the processing relation that follows after this prefix, if this exists. Note that a test function is not uniquely defined, there may be many test functions of the same X-machine.

Example 4. For the X-machine in Example 1, a test function for the sequence $if_1ggf_0f_bhh$ can be constructed as follows:

$t(i) = x_1$ $t(if_1) = x_1x_2$ $t(if_1g) = x_1x_2x_3$ $t(if_1gg) = x_1x_2x_3x_4$ $t(if_1ggf_0) = x_1x_2x_3x_4x_5$ $t(if_1ggf_0f_b) = x_1x_2x_3x_4x_5x_6$ $t(if_1ggf_0f_bh) = x_1x_2x_3x_4x_5x_6x_7$

- $t(if_1ggf_0f_bhh) = x_1x_2x_3x_4x_5x_6x_7$

where $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ are those in Example 3. The construction is correct since $x_1 \in d(i)$, $x_2 \in d(f_1), x_3 \in d(g), x_4 \in d(g), x_5 \in d(f_0), x_6 \in d(f_b), x_7 \in d(h)$, $if_1ggf_0f_b$ is a path from q_0 but $if_1ggf_0f_bh$ is not a path from q_0.

Definition 14. *A test function t^P is called n* natural, *$n \geq 2$, if for any $f_1, \dots, f_p \in \Phi$, $x_1, \dots, x_p \in X$ for all $1 \leq p \leq n$, if $t^P(f_1 \dots f_p) = x_1 \dots x_p$ then $x_i f_i x_{i+1}$ for all $1 \leq i \leq p-1$.*

That is, a test function is n natural when, for any sequence of basic processing relations of length at most n, the values produced by the test function are chosen such that the next value is obtained from the current value through the application of the corresponding processing relation. Example 4 shows the construction of a 4 natural test function which is not 5 natural (since $x_5 \notin g(x_4)$).

Theorem 1. *Let $P = (X, Q, \Phi, F, q_0, d, O, o)$ and $P' = (X, Q', \Phi, F', q_0', d, O, o)$ be two X-machines that are testing-compatible and weak output-distinguishable, let t^P be a test function of P $U \subseteq \Phi^*$ and $I = t^P(U)$. If there exist h_P and $h_{P'}$ two breakpoint computations of P and P', respectively, with $o \circ h_P(x) = o \circ h_{P'}(x)$, $x \in I$ then $A(P)$ and $A(P')$ are U-equivalent.*

Proof. Let $f_1 \dots f_n \in U$. We prove that $f_1 \dots f_n$ is a path in P iff $f_1 \dots f_n$ is a path in P'.

- *if $f_1 \dots f_n$ is a path in P then from Definition 13 it results that there exist $x_i \in dom(f_i)$ for all $1 \leq i \leq n$ such that $t^P(f_1 \dots f_n) = x_1 \dots x_n$ and there exist $y_1, \dots, y_n \in X$ such that $x_i f_i y_i$ for all $1 \leq i \leq n$. According to Definition 9 we may define a breakpoint computation function h_P such that $h_P(x_1 \dots x_n) = y_1 \dots y_n$. If there exists $h_{P'}$ as stated in the hypothesis of this theorem, such that $o \circ h_P(x_1 \dots x_n) = o \circ h_{P'}(x_1 \dots x_n)$ then it follows that there exist $y_1', \dots, y_n' \in X$ such that $o \circ h_{P'}(x_1 \dots x_n) = y_1' \dots y_n'$. Consequently there exist $f_1', \dots, f_n' \in \Phi$ with $x_i f_i' y_i'$ for all $1 \leq i \leq n$, and $o(y_1 \dots y_n) = o(y_1' \dots y_n')$. Since Φ is weak output-distinguishable and $o(y_i) = o(y_i')$, by induction on $1 \leq i \leq n$, it follows that $f_i = f_i'$ and hence $f_1 \dots f_n$ is a path in P'.*

– *if* $f_1 \dots f_n$ *is not a path in* P *then let* $k \in \{0, \dots, n-1\}$ *be the largest number for which* $f_1 \dots f_k$ *is a path in* P. *Then according to Definition 13 we have* $t^P(f_1 \dots f_n) = x_1 \dots x_k x_{k+1}$ *for some* $x_i \in X$, $1 \leq i \leq k+1$. *Let us assume that* $f_1 \dots f_{k+1}$ *is a path in* P'. *Then there exist* $y_1, \dots, y_{k+1} \in X$ *such that* $x_i f_i y_i$, $x_i \in d(f_i)$, *for all* $1 \leq i \leq k+1$ *and* $h_{P'}$ *with* $h_{P'}(x_1 \dots x_{k+1}) = y_1 \dots y_{k+1}$. *Then there exist* $f'_1, \dots, f'_{k+1} \in \Phi, y'_1, \dots, y'_{k+1} \in X$ *such that* $f'_1 \dots f'_{k+1}$ *is a path in* P *and* $x_i f'_i y'_i$ *for all* $1 \leq i \leq k+1$ *and* $o(y_1 \dots y_{k+1}) = o(y'_1 \dots y'_{k+1})$. *Hence* $o(y_i) = o(y'_i)$ *for all* $1 \leq i \leq k+1$. *Since* Φ *is weak output-distinguishable, by induction on* $1 \leq i \leq k+1$, *it follows that* $f_i = f'_i$ *and hence* $f_1 \dots f_{k+1}$ *is a path in* P *which contradicts our assumption. Thus* $f_1 \dots f_{k+1}$ *is not a path in* P'.

Corollary 1. *Let* $P = (X, Q, \Phi, F, q_0, d, O, o)$ *and* $P' = (X, Q', \Phi, F', q'_0, d, O, o)$ *be two X-machines that are testing-compatible and weak output-distinguishable and let* t^P *be a test function of* P. *If* U *is a test set of* $A(P)$ *and* $A(P')$ *such that exist* h_P, $h_{P'}$ *two breakpoint computations of* P *and* P', *respectively, with* $o \circ h_P(x) = o \circ h_{P'}(x)$, $x \in t^P(U)$, *then* $t^P(U)$ *is a breakpoint test set of* P *and* P'.

Proof. Indeed from Theorem 1 it follows that $A(P)$ *and* $A(P')$ *are* U*-equivalent and according to [21] it follows that they are equivalent and consequently* $f^P = f^{P'}$.

Thus, Corollary 1 can be used to generate a breakpoint test set of P and P'; this is $I = t^P(U)$, with U a test set of $A(P)$ [10], [34], [14].

Note that the test set defined in this section consists of a number of sequences of data set elements. Each such sequence exercises a path of the machine and each element of the sequence is applied to the processing relation that labels the corresponding arc, regardless of the result computed by the previous arc. That is, the application of the test set happens as though after processing an arc the machine stops and receives a new data value from the test set. We call this the *breakpoint* testing strategy for X-machines.

Although not unusual in practice, the need to place breakpoints after each arc of the machine is processed will obviously complicate the testing process. Ideally, only the initial value of the data set will have to be supplied to the machine, the subsequent values will be computed by the processing relations that make up the path followed by the machine (this is also the idea behind the concept of an n *natural* test function defined above). This may not always be possible, since in general not all the paths of the machine can be exercised by appropriate initial values of the data set.

However, if a value can be obtained by applying the appropriate processing relation to the previous value in the sequence then there is no need to supply that value to the machine, since it will be simply computed by it. Thus the sequence needs not contain that value, and this can be replaced by a symbol that indicates this. This idea leads to the concept of *extended test set* as presented next.

5 The extended test set of X-machines

Let us first consider some notations and definitions. If $e \notin X$ and X_e denotes $X \cup \{e\}$, then for $x \in X_e$, $y \in X$ we denote by $nvl_e(x, y)$ either x, if $x \neq e$ or y, if $x = e$.

An extended breakpoint computation will be further on considered as a natural extension of breakpoint computation h_P, introduced by Definition 9.

Definition 15. *For an X-machine* P, *a partial function* $\overline{h}_P : XX_e^* \longrightarrow X^*$ *called* an extended breakpoint computation of P *is defined as* $\overline{h}_P(x_1 \dots x_n) = y_1 \dots y_n$, *if there exists a path* $f_1 \dots f_n$ *from* q_0 *such that* $x_1 f_1 y_1$, $x_1 \in d(f_1)$, *and* $nvl_e(x_i, y_{i-1}) f_i y_i$, $nvl_e(x_i, y_{i-1}) \in d(f_i)$, $2 \leq i \leq n$, $n \geq 0$.

Like h_P defined in the previous section, for a non-deterministic X-machine, the partial functions $\overline{h}_P$ are in general not uniquely defined.

Example 5. For P as in Example 1, x_i, y_i, $1 \leq i \leq 7$, y'_4, and a path $u = if_1 g g f_0 f_b f_1$ as in Example 3, if $x_e = x_1 eee x_5 ee$, we may contruct $\overline{h}_P, \overline{h'}_P : X_e X^* \longrightarrow X^*$, such that $\overline{h}_P(x_e) = y$ and $\overline{h'}_P(x_e) = y'$, where $y = y_1 y_2 y_3 y_4 y_5 y_6 y_7$, $y' = y_1 y_2 y_3 y'_4 y_5 y_6 y_7$.

The breakpoint test introduced in Definition 11 may be in a natural way extended in this context by an *extended breakpoint test set* $I_e \subseteq XX_e^*$ by replacing the partial functions $h_P, h_{P'}$ with $\overline{h}_P$ and $\overline{h}_{P'}$ respectively. A test set I_e that is a subset of $X\{e\}^*$ will be called a *natural test set.*

That is, a natural extended breakpoint test set is made up of sequences whose all elements, except the first, are e. This corresponds to the situation where only the initial data value of each sequence is supplied to the two machines.

Lemma 2. *If $I \in X^*$ is a breakpoint test set of P and P' then I is also an extended test set of P and P'.*

Proof. Follows directly from the fact that $I_e = I$ and $\overline{h}_P = h_P$.

Definition 16. *A partial function $E^P : X^* \times \Phi^* \longrightarrow XX_e^*$ is called an* input extention *of P if for any $x_1, \ldots, x_n \in X$, $f_1, \ldots, f_k \in \Phi$, $k+1 \geq n \geq 0$, $E^P(x_1 \ldots x_n, f_1 \ldots f_k) = x_{1,e} \ldots x_{n,e} \in XX_e^*$ as follows:*

- $x_{1,e} = x_1$
- *for $1 < j \leq n$*
  - *if $f_1 \ldots f_{j-1}$ is a path of P that starts at q_0 and $x_{j-1} \in d(f_{j-1})$ then $x_{j,e} = e$ if $x_{j-1} f_{j-1} x_j$ and $x_{j,e} = x_j$ otherwise;*
  - *else $x_{i,e} = x_i$, $j \leq i \leq n$.*

That is, if x_j can be computed by applying the corresponding processing relation to x_{j-1} then $x_{j,e} = e$, otherwise $x_{j,e} = x_j$. Thus E^P provides values different from e only when the corresponding processing relations cannot be exercised by using the previous computed values.

Note that, in general, for a non-deterministic stream X-machine E^P may not be uniquely defined. Also, for $U \subseteq \Phi^*$, $I_e = \cup_{u \in U} E^P(t^P(u), u)$ is a set included in XX_e^*, so an extended computation $\overline{h}_{I_e}^P$ may be defined.

Example 6. For P, x, y as in Example 3 and u, x_e as in Example 5, we may observe that $t^P(u) = x$, $E^P(x, u) = x_e$ and exist $h_P, \overline{h}_P$ such that $h_P(x) = \overline{h}_P(x_e)$ i.e. $h_P(t^P(u)) = \overline{h}_P(E^P(t^P(u), u))$. This is a more general result as shown next.

Lemma 3. *Given an X-machine $P = (X, Q, \Phi, F, q_0, d, O, o)$, t^P a test function of P and $U \subseteq \Phi^*$, for any $\overline{h}_P$ there exists h_P with $\overline{h}_P(E^P(t^P(u), u)) = h_P(t^P(u))$ for all $u \in U$.*

Proof. Let $u = f_1 \ldots f_n \in U$, $x_1 \ldots x_k = t^P(u)$, $x_{1,e} \ldots x_{k,e} = E^P(t^P(u), u)$ and $\overline{h}_P$ an arbitrary extended breakpoint computation such that $\overline{h}_P(x_{1,e} \ldots x_{k,e}) = y_1 \ldots y_k$. According to $\overline{h}_P$ definition we have $x_{1,e} \in d(f_1)$, $x_{1,e} f_1 y_1$ and $nvl_e(x_{i,e}, y_{i-1}) \in d(f_i)$, $nvl_e(x_{i,e}, y_{i-1}) f_i y_i$ for all $2 \leq i \leq k$. We prove now that $x_i \in d(f_i), x_i f_i y_i$ for all $1 \leq i \leq k$. We prove this by induction on $1 \leq i \leq k$. This is obviously true for $i = 1$ since $x_{1,e} = x_1$. We assume that the property holds for $1 \leq j \leq i$. If $x_{i+1,e} \neq e$ then $x_{i+1,e} = x_{i+1}$ and the property holds for $j+1$ as $nvl_e(x_{i+1}, y_i) = x_{x+1}$. Otherwise we have $x_{i+1} = y_i$, so $y_i \in d(f_{i+1}), y_i f_{i+1} y_{i+1}$ and again the property holds for $j+1$. Hence we may define a breakpoint computation h_P such that $h_P(x_1 \ldots x_k) = y_1 \ldots y_k$, and according to the construction above has the property $\overline{h}_P(E^P(t^P(u), u)) = h_P(t^P(u))$ for all $u \in U$.

Theorem 2. *Let $P = (X, Q, \Phi, F, q_0, d, O, o)$ and $P' = (X, Q', \Phi, F', q_0', d, O, o)$ be two X-machines that are testing-compatible and strong output-distinguishable and let t^P be a test function of P, $U \subseteq \Phi^*$, E^P an input extension of P and $I_e = \cup_{u \in U} E^P(t^P(u), u)$. If there exist $\overline{h}_P$ and $\overline{h}_{P'}$ two extended breakpoint computations of P and P', respectively with $o \circ \overline{h}_P(x_e) = o \circ \overline{h}_{P'}(x_e)$, $x_e \in I_e$ then $A(P)$ and $A(P')$ are U-equivalent.*

Proof. We prove that if $o \circ \overline{h}_P(x_e) = o \circ \overline{h}_{P'}(x_e)$ then there exist h_P and $h_{P'}$ with $o \circ h_P = o \circ h_{P'}$, on I, where $I = t^P(U)$. The result we are after will then follow from this assertion and Theorem 1.

Let Pt_0 be the set of all paths of P that start at q_0 and Pt_0' the set of all paths of P' that start at q_0'. Let $u = f_1 \ldots f_n \in U$, $x = x_1 \ldots x_k = t^P(u)$, $x_e = x_{1,e} \ldots x_{k,e} = E^P(t^P(u), u)$, $k \leq n$. Then it follows that an extended breakpoint computation $\overline{h}_P$ exists such that $\overline{h}_P(x_e) = y = y_1 \ldots y_k$. Then from Lemma 3 there exists h_P with $h_P(x) = \overline{h}_P(x_e) = y$.

From $o \circ \overline{h}_P = o \circ \overline{h}_{P'}$, over I_e it follows that $y = y_1 \dots y_k \in h_P(x)$, $y' = y'_1 \dots y'_k \in \overline{h}_{P'}(x_e)$ with $o(y) = o(y')$.

Thus $f_1 \dots f_k \in Pt_0$ and $x_i \in d(f_i), x_i f_i y_i$ for all $1 \leq i \leq k$ if and only if there exists $f'_1 \dots f'_k \in Pt'_0$ with $x_{1,e} \in d(f'_1), x_{1,e} f'_1 y'_1$ and $nvl_e(x_{i,e}, y'_{i-1}) \in d(f'_i), nvl_e(x_{i,e}, y'_{i-1}) f'_i y'_i$ for all $2 \leq i \leq k$. From $o(y_i) = o(y'_i)$ for all $1 \leq i \leq k$, and using the strong output-distinguishability property of P, by induction on $1 \leq i \leq k$ it follows that $f_i = f'_i$ for all $1 \leq i \leq k$ and $x_i = nvl_e(x_{i,e}, y'_{i-1})$ for all $2 \leq i \leq k$.

Thus $f_1 \dots f_k \in Pt_0$ and $x_i \in d(f_i), x_i f_i y_i$ for all $1 \leq i \leq k$ if and only if $f'_1 \dots f'_k \in Pt'_0$ and $x_i \in d(f'_i), x_i f'_i y'_i$ for all $1 \leq i \leq k$, and $o(y) = o(y')$. Hence we may build $h_{P'}$ such that $o \circ h_P(x) = o \circ h_{P'}(x)$, $x \in I$. Applying now Theorem 1 it follows that $A(P)$ and $A(P')$ are $U-$equivalent.

Corollary 2. *Let $P = (X, Q, \Phi, F, q_0, d, O, o)$ and $P' = (X, Q', \Phi, F', q'_0, d, O, o)$ be two X-machines that are testing-compatible and strong output-distinguishable and let t_P be a test function of P such that exist $\overline{h}_P$, $\overline{h}_{P'}$ two extended breakpoint computations of P and P', respectively, with $o \circ \overline{h}_P(x) = o \circ \overline{h}_{P'}(x)$, $x \in t^P(U)$, and E^P an input extension of P. If $U \subseteq \Phi^*$ is a test set of $A(P)$ and $A(P')$ then and $I_e = \cup_{u \in U} E^P(t^P(u), u)$ is an extended breakpoint test set of P and P'.*

Proof. See Corollary 1.

Theorem 3. *Let $P = (X, Q, \Phi, F, q_0, d, o)$ and $P' = (X, Q', \Phi, F', q'_0, d, o)$ be two X-machines that are testing-compatible and strong output-distinguishable. If there exists t^P an n natural test function of P for n sufficiently large, then there exists a natural extended breakpoint test set I_e of P and P'.*

Proof. Let U be a test set of $A(P)$ and $A(P')$, n the length of the longest sequence in U and t an n natural test function of P. From Corollary 2 it follows that $I_e = \cup_{u \in U} E^P(t^P(u), u)$ is an extended breakpoint test set of P and P'. Since t^P is n natural, from Definition 16 it follows that $I_e \subseteq X\{e\}^*$.

That is, the existence of an n natural test function for n sufficiently large will ensure that any path in U can be exercised using as initial data values the head elements of the sequences in I_e. Thus, I_e is a natural extended breakpoint test set of P and P'.

6 Stream X-machines and stream X-machine testing

This section introduces the stream X-machine and particularises the results given in the previous sections to this class of X-machines.

Definition 17. *An X-machine $P = (X, Q, \Phi, F, q_0, d, \Gamma, o)$ is called a* stream X-machine *(SXM for short) if the following are true:*

- *$X = \Gamma^* \times M \times \Sigma^*$, where M is the (possibly infinite)* memory*, Σ is the finite* input alphabet *and Γ is the finite* output alphabet*;*
- *if $f \in \Phi$ then $dom(f) \cap (\Gamma^* \times M \times \{\lambda\}) = \emptyset$ and there exists a relation $\phi_f : M \times \Sigma \longleftrightarrow \Gamma \times M$ such that $\forall g \in \Gamma^*, m, m' \in M, s \in \Sigma^*, \gamma \in \Gamma, \sigma \in \Sigma$ $(g, m, s\sigma) f (g\gamma, m', s)$ if and only if $(m, \sigma)\phi_f(\gamma, m')$;*
- *$\forall g \in \Gamma^*, m \in M$, $c(g, m, s) = true$ if and only if $s = \lambda$;*
- *o is a function $o : \Gamma^+ \times M \times \Sigma^* \longrightarrow \Gamma$ defined by $o(g, m, s) = rear(g), g \in \Gamma^*, m \in M, s \in \Sigma^*$ (o returns the rightmost element of any non-empty output sequence).*

The output set O used in the definition of an X-machine is now Γ in the above definition. Please note that Γ is a finite set.

An SXM may have state non-determinism, operator non-determinism and domain non-determinism but not termination non-determinism. As for the general X-machine model, only operator and domain non-determinism will be considered in what follows.

Note 6. For simplicity, in what follows we will consider processing relations of the form $f : M \times \Sigma \longleftrightarrow \Gamma \times M$ (i.e. we will use ϕ_f instead of f from Definition 17). Similarly, we will consider that the testing domain of the machine is a function $d : \Phi \longrightarrow 2^{M \times \Sigma}$. An SXM will be denoted by a tuple $P = (\Sigma, \Gamma, M, \Phi, F, q_0, d)$.

Note 7. An SXM is weak output-distinguishable if for any $f, g \in \Phi$, if there exist $m, m'_1, m'_2 \in M, \sigma \in \Sigma, \gamma \in \Gamma$ with $(m, \sigma)f(\gamma, m'_1)$, $(m, \sigma)g(\gamma, m'_2)$ then $f = g$.

Note 8. Two stream X-machines are testing compatible if their input alphabets, output alphabets, memory sets, sets of processing relations and testing domains coincide.

Example 7. Even though the X-machine in Example 1 processes a sequence of input symbols, it is not an SXM (i.e. i does not process the input sequence in an SXM fashion and the data set does not include explicitly an output alphabet). However, it can be easily rewritten in this fashion, as shown next. So $X = \Gamma^* \times M \times \Sigma^*$ with $\Sigma = \{c \mid c \text{ is letter or digit }\} \cup \{@, \$\}$ ($\$$ is the blank character and @ is a symbol used to mark the start of the input sequence), $M = N \times N$, $\Gamma = \{\sigma_f \mid \sigma \in \Sigma, f \in \Phi\}$, $\Phi = \{i, e, f_0, f_1, f_b, g, h\}$, where the definitions of the processing relations (written in the fashion of Note 6) are as follows:

$((0,0), @)i(@_i, (0,0))$,
$((0,k), c)e(c_e, (0,k))$, where $k \in N$, $c \in \Sigma \setminus \{1, @, \$\}$,
$((0,k), 0)f_0(0_{f_0}, (0, k-1))$, where $k \in N, k > 0$,
$((0,k), 1)f_1(1_{f_1}, (1, k+1))$, where $k \in N$,
$((n,k), b)f_b(b_{f_b}, (0,k))$, where $k, n \in N$,
$((n,k), x)g(x_g, (n+y, k))$, where $k, n \in N$, ($n > 0$, $x \in \{0, c\}$ and $y = -1$) or ($x \in \{1, c\}$ and $y = 1$), where $c \in \Sigma \setminus \{0, 1, @, \$\}$,
$((n,k), x)h(x_h, (n,k))$, where $k, n \in N$, $x \in \Sigma \setminus \{@\}$.

The definition of F remains unchanged.

Please note that the values $\sigma_f (\sigma \in \Sigma, f \in \Phi,)$ are considered in Γ in order to keep the machine output-distinguishable. Also note that the "dummy" values $-1, -2, -3, -4$ used in example 1 are no longer needed and have been removed.

Definition 18. *For an SXM P, a partial function $h'_P : (M \times \Sigma)^* \longrightarrow \Gamma^*$ called an* input-memory computation *of P is defined as $h'_P(x_1 \dots x_n) = \gamma_1 \dots \gamma_n$, if there exists a path $f_1 \dots f_n$ from q_0 and $m_i \in M$, $1 \le i \le n$, such that $x_i \in d(f_i)$, $x_i f_i(\gamma_i, m_i)$, $1 \le i \le n$, $n \ge 0$.*

Note 9. The test function introduced by Definition 13 will be

- $t^P(\lambda) = \lambda$;
- for $p \ge 0$, consider t^P defined for any string $f_1 \dots f_p \in \Phi^*$; then for any string $f_1 \dots f_p f_{p+1}$, $t^P(f_1 \dots f_p f_{p+1})$ is either of:
  - if $f_1 \dots f_p$ is a path in P that emerges from the initial state q_0 then $t^P(f_1 \dots f_p f_{p+1}) = t^P(f_1 \dots f_p)x_{p+1}$, where $x_{p+1} \in d(f_{p+1})$, with $x_{p+1} = (m, \sigma)$, $m \in M$, and $\sigma \in \Sigma$.
  - otherwise $t^P(f_1 \dots f_p f_{p+1}) = t^P(f_1 \dots f_p)$.

In this case $t^P(U)$, $U \subseteq \Phi^*$ contains sequences of pairs $(m, \sigma) \in M \times \Sigma$.

Example 8. For P as in Example 7, with $u = i f_1 g g f_0 f_b f_1$ and $z = z_1 z_2 z_3 z_4 z_5 z_6 z_7$, where

$z_1 = ((0,0), \$)$, $z_2 = ((0,0), 1)$, $z_3 = ((1,1), 1)$, $z_4 = ((2,1), b)$, $z_5 = ((0,1), 0)$, $z_6 = ((0,0), b)$, $z_7 = ((0,0), 1)$.

we can construct h'_P such that $h'_P(z) = \gamma$, where $\gamma = @_i 1_{f_1} 1_g b_g 0_{f_0} \$_{f_b} 1_{f_1}$.

Definition 19. *Let $P = (\Sigma, \Gamma, M, Q, \Phi, F, q_0, d)$ and $P' = (\Sigma, \Gamma, M, Q', \Phi, F', q'_0, d)$ be two stream X-machines that are testing-compatible. Then a finite set $I \subseteq (M \times \Sigma)^*$ is called an* input-memory test set *of P and P' if whenever there exist h'_P and $h'_{P'}$ two input-memory computations of P and P', respectively, with $h'_P(x) = h'_{P'}(x)$, $x \in I$, we have $f^P = f^{P'}$.*

Theorem 4. *Let $P = (\Sigma, \Gamma, M, Q, \Phi, F, q_0, d)$ and $P' = (\Sigma, \Gamma, M, Q', \Phi, F', q_0', d)$ be two stream X-machines that are testing-compatible and weak output-distinguishable and let t^P be a test function of P. If $U \subseteq \Phi^*$ is a test set of $A(P)$ and $A(P')$ such that exist h'_P and $h'_{P'}$ two input-memory computations of P and P', respectively, with $h'_P(x) = h'_{P'}(x)$, $x \in t^P(U)$, then $t^P(U)$ is an input-memory test set of P and P'.*

Proof. Follows directly from Theorem 1 and Definition 18.

Definition 20. *For an SXM P, a partial function $h'_{P,m} : \Sigma^* \longrightarrow \Gamma^*$ called an* input computation *of P in $m \in M$ is defined as $h'_{P,m}(\sigma_1 \dots \sigma_n) = \gamma_1 \dots \gamma_n$, if there exists a path $f_1 \dots f_n$ from q_0 and $m_i, m_i' \in M$, $1 \leq i \leq n-1$, $m_0, m_n' \in M$, $m_0 = m$, such that $(m_{i-1}, \sigma_i) \in d(f_i)$, $(m_{i-1}, \sigma_i) f_i(\gamma_i, m_i')$, $1 \leq i \leq n$, $n \geq 0$.*

Note 10. The test function t^P will be defined similar to Note 9 where the definition of $t^P(f_1 \dots f_p f_{p+1})$ when $f_1 \dots f_p$ is a path in P will be replaced by $t^P(f_1 \dots f_p f_{p+1}) = t^P(f_1 \dots f_p)\sigma_{p+1}$, $\sigma_{p+1} \in \Sigma$, if exists $m_{p+1} \in M$ such that $(m_{p+1}, \sigma_{p+1}) \in d(f_{p+1})$. In this case $t^P(U)$, $U \subseteq \Phi^*$ contains sequences of input symbols $\sigma \in \Sigma$.

Definition 21. *Let $P = (\Sigma, \Gamma, M, Q, \Phi, F, q_0, d)$ and $P' = (\Sigma, \Gamma, M, Q', \Phi, F', q_0', d)$ be two stream X-machines that are testing-compatible. Then a finite set $I \subseteq \Sigma^*$ is called an* input test set *of P and P' if whenever there exist $m \in M$, $h'_{P,m}$ and $h'_{P',m}$ two input computations of P and P', respectively, in m with $h'_{P,m}(x) = h'_{P',m}(x)$, $x \in \Sigma^*$, we have $f^P = f^{P'}$.*

Theorem 5. *Let $P = (\Sigma, \Gamma, M, Q, \Phi, F, q_0, d)$ and $P' = (\Sigma, \Gamma, M, Q', \Phi, F', q_0', d)$ be two stream X-machines that are testing-compatible, weak output-distinguishable and let t^P be a test function of P. If $U \subseteq \Phi^*$ is a test set of $A(P)$ and $A(P')$ such that exist $m \in M$ and $h'_{P,m}$, $h'_{P',m}$ two input computations of P and P', respectively, in m with $h'_{P,m}(x) = h'_{P',m}(x)$, $x \in t^P(U)$, then $t^P(U)$ is an input test set of P and P'.*

Proof. It follows directly from Theorem 1 and Definition 20.

Note 11. Similar to n natural test function we may allow a slightly modified condition which says that if $f_1, \dots, f_n \in \Phi$ and exist $\sigma_i \in \Sigma$, $\gamma_i \in \Gamma$, $m_i \in M$, $1 \leq i \leq n$, $m_0 \in M$ with $(m_{i-1}, \sigma_i) f_i(\gamma_i, m_i)$, $1 \leq i \leq n$, then $t^P(f_1 \dots f_n) = \sigma_1 \dots \sigma_n$.

In this case we recapture the test function definition used in [21] as an weak n natural test function with n greater than or equal to the length of the longest sequence of U.

7 Conclusions

The paper provides the theoretical basis for the generation of test sets from non-deterministic X-machine specifications. A *test set* is a finite set of sequences of elements that guarantees that two machines have identical behaviour when the application of this set to the two machines produces identical results. In fact, the implication is stronger: not only the behaviours of the two machines coincide, but also their associated finite state machines. Thus the approach is to generate sequences of arcs that can test this equivalence (using the existing testing theory for finite state machines) and to exercise these using appropriate sequences of elements in the data set. Two approaches are investigated. The first is to supply each arc in a path with a suitable value of the date set and this corresponds in practice to placing breakpoints after each relation has been processed. This gives rise to the *breakpoint test set.* The *extended test set* strategy eliminates the breakpoints whenever this is possible by reusing values from the previous computation. Ideally, only the first value in the sequence would be supplied, this corresponds to the case when an n *natural test function* can be constructed for a sufficiently large n.

These two approaches are then particularised to the stream X-machine class and the conditions in which a stream X-machine admits an n natural test function are identified. The theoretical results in this paper are generalisations of the results in [26].

8 Acknowledgements

We would like to thank the anonymous reviewers for their very helpful comments, allowing us to improve the quality of this paper. We are also garteful to the EPSRC projects MOTIVE (grant GR/M56777) and FORTEST (GR/R43150) for partially funding our work.

References

1. Aho, A.V., Dahbura, A.T., Lee, D. and Uyar, M.U. (1988) An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. Proceedings of Protocol Specification, Testing, and Verification VIII, 75-86, Atlantic City, North-Holland.
2. T. Bălănescu, Generalised stream X-machines with output delimited type, *Formal Aspects of Computing*, 12, 473-484, 2000.
3. T. Bălănescu, M. Gheorghe, M. Holcombe, Deterministic stream X-machines based on grammar systems, in *Words, Sequences, Grammars, Languages: where Biology, Computer Science, Linguistics and Mathematics meet,* volume 1, C. Martin-Vide and V. Mitrana eds., Kluwer, 2000.
4. J. Barnard, J. Whitworth, M. Woodward, Communicating X-machines, *Information and Software Technology,* 38, 401-407, 1996.
5. F. Belina, D. Hogrefe, The CCITT-specification and description language, *Computer Networks and ISDN Systems,* 16, 1347-1356, 1989.
6. G. Bernot, M. Gaudel and B. Marre, Software testing based on formal specifications: a theory and a tool, *Software Engineering Journal*, 6, 387-405, 1991.
7. K. Bogdanov, Automated testing of Harel's statecharts, PhD thesis, Sheffield University, UK, 2000.
8. T. Bolognesi, E. Brinksma, Introduction to the ISO specification LOTOS, *Computer Networks and ISDN Systems,* 14(1), 25-59, 1987.
9. K.-T. Cheng and A.S. Krishnakumar, Automatic functional test generation using the extended finite state machine mode, *Proc. DAC*, 1-6, 1993.
10. T. S. Chow, Testing software design modelled by finite state machines, *IEEE Transactions on Software Engineering,* 4, 178-187, 1978.
11. P. Dauchy, M Gaudel and B. Marre, Using algebraic specifications in software testing: a case study on the software of an automatic subway, *Journal of Systems Software*, 21, 229-244, 1993.
12. S. Eilenberg, *Automata, Languages and Machines,* Vol. A, Academic Press, 1974.
13. M. Fairtlough, M. Holcombe, F. Ipate, C. Jordan, G. Laycock and Z. Duan, Using an X-machine to model a video cassette recorder, *Current Issues in Electronic Modelling,* 3, 141-161, 1995.
14. S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou and A. Ghedamsi, Test selection based on finite state models, *IEEE Transactions on Software Engineering,* 17, 591-603, 1991.
15. G. Gonenc, A method for the design of fault detection experiments, *IEEE Transactions on Computer,* 19, 551-558, 1970.
16. R.M. Hierons, Extending test sequence overlap by invertibility, *The Computer Journal,* 39, 325-330, 1996 .
17. R.M. Hierons, Testing from a finite state machine: extending invertibility to sequences, *The Computer Journal,* 40, 220-230, 1997.
18. R.M. Hierons, M. Harman, Testing conformane to a quasi-non-deterministic stream X-machine, *Formal Aspects of Computing,* 12, 423-442, 2000.
19. M. Holcombe, X-machines as a basis for dynamic system specification, *Software Engineering Journal,* 3, 69-76, 1988.
20. M. Holcombe, F. Ipate and A. Grondoudis, Complete functional testing of safety-critical systems, *Safety and Reliability in Emerging Control Technologies: A Postprint volume from the IFAC Workshop on Safety and Reliability in Emerging Control Technologies*, Daytona Beach, Florida, USA, 199-204, 1-3 November 1995.
21. M. Holcombe and F. Ipate, *Correct Systems: Building a Business Process Solution,* Springer Verlag, Berlin, 1998.
22. F. Ipate and M. Holcombe, Another look at computability, *Informatica,* 20, 359-372, 1996.
23. F. Ipate and M. Holcombe, An integration testing method that is proved to find all faults, *International Journal of Computer Mathematics,* 63, 159-178, 1997.
24. F. Ipate and M. Holcombe, Specification and testing using generalized machines: a presentation and a case study, *Software Testing, Verification and Reliability,* 8, 61-81, 1998.
25. F. Ipate and M. Holcombe, A method for refining and testing generalised machine specifications, *International Journal of Computer Mathematics,* 68, 197-219, 1998.

26. F. Ipate and M. Holcombe, Generating test sequences from non-deterministic generalised stream X-machines, *Formal Aspects of Computing,* 12, 443-458, 2000.
27. F. Ipate, M. Gheorghe and M. Holcombe, Testing (stream) X-machines, submitted.
28. Z. Kohavi, *Switching and Finite Automata Theory,* McGraw-Hill Book Company, 1978.
29. D. Lee and M. Yannakakis, Principles and methods of testing finite state machines - A survey, *Proceedings of the IEEE*, 84, 1090-1123, 1996.
30. G. Luo, G. v. Bochmann and A. Petrenko, Test selection based on communicating non-deterministic finite-state machines using a generalised Wp-method, *IEEE Transactions on Software Engineering,* 20, 149-161, 1994.
31. S. Naito and M. Tsunoyama, Fault detection for sequential machines, *Proc IEEE Fault Tolerant Comput. Conf.,* 1981.
32. A. Petrenko, Fault model-driven test derivation from finite state models: annotated bibliography, in *Modelling and Verification of Parallel Processes,* F. Cassez, C. Jard, B. Rozoy, M.D. Ryan eds., LNCS 2067, 196-205, 2001.
33. K.K. Sabnani and A.T. Dabhura, A protocol testing procedure. *Computer Networks and ISDN Systems,* 5, 285-297, 1988.
34. D.P. Sidhu and T.-K. Leung, Formal methods for protocol testing: a detailed study, *IEEE Transactions on Software Engineering,* 15, 413-426, 1989.
35. Yang, B. and Ural, H. Protocol conformance test generation using multiple UIO sequences and overlapping, *ACM SIGCOMM 90: Communications, Architectures and Protocols,* Twente, The Netherlands, September 24-27, 118-127, 1990.

Complete Behavioural Testing
(two extensions to state-machine testing)

Mike Stannett
MOTIVE Research Associate
Verification and Testing Research Group
Dept of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom.
m.stannett@dcs.shef.ac.uk

Abstract. We present two extensions to state-machine based testing. The first allows us to extend Chow's *W*-method (CHOW) to machines with non-final states, thereby including infinitely many more specifications within the scope of the method. The second is a radical extension to Holcombe and Ipate's 1998 stream *X*-machine testing method (SXMT), itself an extension of CHOW. This method involves the construction of a bespoke model of computation, the *behavioural machine* (*β–machine*), which extends Eilenberg's 1974 *X*-machine concept. Examples of *β*–machines include *X*-machines, stream *X*-machines, semi-automata and automata, and even some concurrent system specifications. Despite this generality, we demonstrate an algorithm for generating complete behavioural test sets for systems specified by *β*–machines.

Keywords. Algebraic models of computation, Chow's W-method, complete functional testing, concurrent systems, object-oriented testing, software testing, stream X-machine (SXM).

Acknowledgement. This research is sponsored by the UK-EPSRC through the project, *MOTIVE* (GR/M56777: *Method for Object Testing, Integration and Verification*; principal investigator: Dr Anthony J. Simons; co-investigator: Prof W. Mike L. Holcombe).

1 Introduction

This paper describes a unified approach to testing sequential and concurrent systems, which extends both Chow's *W*-method (CHOW) for verifying designs presented as state machines [Cho78], and Holcombe and Ipate's SXMT test-generation method [HI98] for the stream *X*-machine (SXM) [Lay93]. In addition to covering more classes of specification model, our methods also allow the coverage of more machines within each of the standard classes. Both CHOW and SXMT require the system specification to be a minimal completely specified machine, in which every state is final and precisely one state is initial, and these constraints ensure that infinitely many specifications are untestable within the method. In contrast, we present a simple technique, *super-minimisation*, that allows machines with non-final states to be put into a form that satisfies the CHOW- and SXMT-style preconditions. Moreover, super-minimisation results in a machine with up to 50% fewer states than the standard minimised variant of the original semi-automaton. Because the number of tests required to characterise a machine's behaviour depends on the size of the underlying state set, our technique can result in significantly smaller test sets.

Although there are differences between CHOW and SXMT, these are essentially imposed by the different natures of the systems being modelled (automata and stream *X*-machines, respectively). We present a unified model for state-machine-style specifications, which we call the *behavioural machine* (*β*–machine). This model includes many standard sequential models as instances, including automata, stream *X*-machines and standard *X*-machines, but also applies to concurrent models. Moreover, *β*–machines can be equipped with a general purpose test-generation method of which CHOW and SXMT are instances. Just as SXMT can be used to

generate complete functional tests of sequential systems, so the β–machine method (β–method) can be used to generate complete behavioural tests for β–machines of any sort.

1.1 Structure of the paper

In Section 2, we illustrate our simple super-minimisation technique for including machines with non-final states within the remit of CHOW and SXMT. In Section 3 we present our unified behavioural model, the *β–machine*, for state-machine-style computations. We show that *X*-machines [Eil74, Hol89, Sta90], automata, and even some concurrent specifications are all types of β–machine. In Section 4, we summarise and extend CHOW and SXMT. We demonstrate an algorithm for generating test-sets for β–machines, and prove that they are behaviourally complete.

1.2 Notation and conventions

Finite state machines can be defined both with and without outputs. Following [LP84], a finite state machine (FSM) without output will be called a *semi-automaton*, and an FSM with output will be called a *automaton*. Throughout this paper, F denotes an FSM with input alphabet A (also denoted In), output alphabet Out (if F is a semi-automaton, we take $Out = In$ and regard each label a as the input/output pair a/a), state set S, initial state set $I \subseteq S$ and final (*aka* terminal) state set $T \subseteq S$. Given any $U \subseteq S$ and $\mathbf{a} \in A^*$, we write $U \Rightarrow^{\mathbf{a}}$ to mean that there exists a path in F, labelled **a,** starting at some $s \in U$. Likewise, $U \Rightarrow^{\mathbf{a}} V$ means there is a path labelled **a** from a state in U to a state in V. If $\mathbf{a} = (a)$ is a string of length one, we write $\rightarrow^{a}$ in place of $\Rightarrow^{\mathbf{a}}$, and if $U = \{s\}$ is a singleton, we write $s \Rightarrow^{\mathbf{a}}$ in place of $\{s\} \Rightarrow^{\mathbf{a}}$. F is *completely specified* (or *complete*) if, for each $s \in S$ and $a \in A$, there is at least one transition $s \rightarrow^{a}$, and *deterministic* if there is at most one such transition. A language $L \subseteq A^*$ is a *(state) cover* for F if, for each state s, there is some $\mathbf{a} \in L$ with $I \Rightarrow^{\mathbf{a}} s$. Given any language $L \subseteq A^*$, and any integer n, we define $L^{(n)} = \cup\{L^k \mid 1 \le k \le n\} \cup \{\varepsilon\}$, where ε is the empty string over A. For $n \le 0$, this reduces to $L^{(n)} = \{\varepsilon\}$.

If F has at least one cover, we say that F is *accessible*. If, given any states $s \neq t$, we can find **a** with $s \Rightarrow^{\mathbf{a}} t$, then F is said to be (*strongly*) *connected*; any connected machine is accessible. Conversely, if an accessible machine F has only one initial state and is equipped with a 'reset' action which returns the system to this state on demand (i.e. if there is a transition $s \rightarrow^{reset} I$ from each state s), then F is strongly connected.

2 Testing semi-automata with non-final states

Consider the simple semi-automaton (FSM without output) shown in *Figure 1a*. We can think of this as the specification of a system which is allowed to perform any combination of *a*s and *b*s, provided the last operation it performs it definitely an *a* (for example, industrial machinery might carry the specification "power must be turned off at the end of the shift"). The figure shows the minimal machine for the relevant language A^*a, where $A=\{a,b\}$. It has two states, of which the one on the right is final and the one on the left initial (but not final).

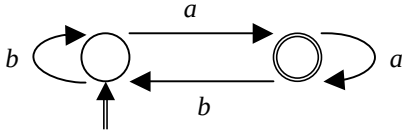


Figure 1a. Minimal semi-automaton for A^*a

We can regard any semi-automaton as an automaton by interpreting each symbol *a* as the input/output pair *a/a*, so it's meaningful to consider this machine in relation to CHOW. In fact, however, neither Chow's *W*-method nor its functional extension, the SXM testing method, can be used to generate tests for systems with this sort of specification, because these test methods require all states in the minimised specification machine to be final, and here we have a non-final state. The minimal versions of infinitely many specifications fail the very stringent conditions of these methods.

To overcome this problem, we need to re-express the specification as a minimal machine in which all states are final. We do this by *super-minimising* the machine. Rather than representing each symbol *a* as the pair *a*/*a*, we instead append a 'fake' output symbol to capture the termination behaviour of the relevant transition. If the transition ends at a final state we add the 'output' T (*true*), and otherwise F (*false*). The initial arrow is labelled ε/F (in this example) to indicate that ε is not deemed a member of the behaviour. The distinction between final and non-final states is now superfluous, because all of the relevant information is encoded by the T/F components of the transition labels (*Figure 1b*). We can therefore declare all states in this 'transducer' representation to be final, without losing any information.

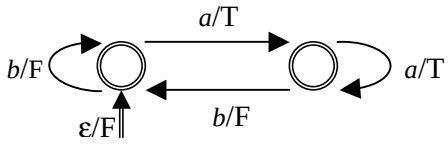


Figure 1b. Generated machine for *A*a*

Because we have imposed a new state configuration, there is scope for further minimisation. Minimising this particular machine yields the minimal machine in *Figure 1c*, with just one state.

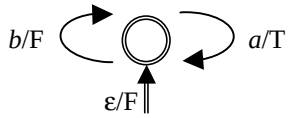


Figure 1c. Super-minimised β–machine for *A*a*.

Consequently, we have achieved a 50% reduction in the state set, even for a very simple machine that was already minimal. This is a significant saving, and has the important corollary that Chow's *W*-method can now be applied to a behaviour that was previously untestable by this method. To see how to use the method, suppose we are given a 2-state implementation to test. In this case the implementation under test (IUT) has one more state than the (super-minimal) specification. We use Figure 1c to choose a cover, say $C = \{\varepsilon\}$, and a characterisation set $W = \{a\}$ and construct the CHOW test-set $TS = CA^{(2)}W$, *i.e.*

$$TS = \{ a, aa, ba, aaa, aba, baa, bba\}$$

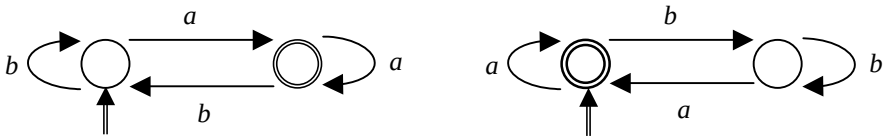
Looking at the specification, we see that the accept/reject responses for each of these strings is

a	T	*aaa*	TTT
aa	TT	*aba*	TFT
ba	FT	*baa*	FTT
		bba	FFT

Have we successfully generated a test set? To find out, suppose *Imp* is a 2–state semi-automaton with the accept/reject patterns specified. Call the states L (left) and R (right), with L initial. It is clear that transitions $L \rightarrow^a$ and $L \rightarrow^b$ must exist, because there are extensions of these 1-

transition paths that are accepted. Moreover, the first clearly leads to an acceptance state and the other to a rejection. There are only two layouts that meet these criteria

Looking at the length-2 patterns in the test set allows us to complete the diagrams (again we use the acceptance of length-3 paths to determine that the second *b* transition must exist)



The languages accepted by these machines differ only as to whether they include ε. We contend that "testing ε" is a meaningless activity, because no test based on the empty input string ε can be assured of generating any observable output. This is why we require the designer to specify explicitly whether or not ε is to be included in the language. In this case we have the specification ε/F, so we reject the right-hand machine.

Clearly, the only 2-state semi-automaton that satisfies our test-set requirements is the machine we started with, so our 2-state-machine test set is indeed working. The reason it works is straightforward: the super-minimal machine precisely captures the accept/reject behaviour of the original semi-automaton. Consequently, any machine constructed to match the behaviour of the super-minimal machine automatically matches that of the specification machine itself. Moreover, we can obviously extend the method to allow testing of more complex machine types. So, for example, we can generate tests for automata or stream *X*-machines with non-final states by extending *Out* to include the T/F information, and likewise, by adding T/F information to the fundamental datatype *X*, we can generate test sets for *X*-machines with non-final states (see below for an explanation of *X*-machines and stream *X*-machines).

One word of caution. We have seen how we can regard semi-automata as automata with T/F outputs, where a transition emits T if and only if it leads to a final state, and emits F otherwise. Chow's method applies to this automaton, and hence to the underlying semi-automaton, but we have to be very careful when applying the method, because the behaviour of the machine *F* is different when viewed as an automaton, as compared with its behaviour when viewed as a semi-automaton. As a semi-automaton, all that matters when we supply an input string **a** is whether or not it is recognised. But as an automaton, we also have the history of acceptance/rejection decisions as the string was submitted. This extra information is vital to Chow's method, so we need to make it available. We accordingly require of any test set that it be *downward closed*. That is, if **a** is in the test set, so is every prefix of *A*. To make this unambiguous, we will write $L{\downarrow}$ to denote the downward-closure of the language *L*. Obviously, if *L* is finite, so is $L{\downarrow}$. The extent to which the need to use downward closures *matters* is open to debate. In the example just considered, for instance, it is clear that the information obtained from the input strings T and TT is also available in the output recorded for the longer string TTT. So while downward closure is necessary as a theoretical precaution, it may have little relevance to practical test generation.

3 Behavioural frameworks and β–machines

Whenever we draw a state-machine diagram, we implicitly make certain assumptions about the symbols used to label the transitions. These labels can vary considerably in algebraic sophistication, but always share certain key properties. To put things in context, let's review the difference between automata, semi-automata, *X*-machines and stream *X*-machines.

All four machine types are based on the standard finite state machine, but differ as to the labels they associate with transitions. Rather than deal with large numbers of definitions, we'll assume instead that *F* is a semi-automaton with alphabet *A*, and that *B* is some set of behavioural labels that we're going to associate with transitions. In other words, each symbol *a* becomes the *name* of a behaviour in *B*, and we can represent this by supposing the existence of some labelling function $\Lambda: A \to B$.

Loosely speaking, the labelling involved in each of the models is as follows. For semi-automata the labelling is the identity map, $\Lambda(a) = a$, and for automata we essentially take $B = In \times Out$. For an *X*-machine (XM), the labelling maps *A* into $R(X)$, the set of all relations of type $X \leftrightarrow X$. For a stream *X*-machine (SXM), we map *A* into a set of relations of type $In \times Mem \leftrightarrow Mem \times Out$, where *Mem* is a potentially infinite auxiliary set called the machine's *memory*.

The basic operations involved in representing languages by semi-automata are *concatenation* and *aggregation*. Concatenation occurs when one transition follows another. Thus, for example, when we regard the two transitions $\rightarrow^{u}\rightarrow^{v}$ as a single *path* $\Rightarrow^{uv}$, we are implicitly assuming that the labels *u* and *v* can be concatenated to yield the string *uv*. Notice that the type of *uv* differs from that of *u* and *v* themselves. This is unacceptable for our purposes, because we want to build a model of behaviour which is self-contained. The concatenation of two behaviours should be another behaviour – it should have the same type. We can overcome the problem easily by thinking of *u* and *v* as the strings (*u*) and (*v*) of length one. Now, behaviours are represented as strings, and the concatenation of the two strings (*u*) and (*v*) is again a string. *Aggregation* occurs when we combine the various successful paths through a machine and regard them as a single *language*. Again, this seems to violate our principle that all behaviours should be of the same type, because we are operating on strings and generating *sets of* strings. Fortunately there is again a simple solution. Wherever we have a string **u**, we regard it as a language {**u**} with a single element. Looked at in this way, aggregation is simple set-theoretic union. Given these interpretations, we see that semi-automata can be modelled by taking a labelling of the form $\Lambda(a) = \{(a)\}$, which replaces each symbol *a* with the singleton language containing the single-character string (*a*). This is important for our purposes because we now have a model of FSM computation in which all components are of the same basic type (in this case $\wp A^*$), and this type is closed under concatenation and aggregation.

More generally, we take *B* to be any algebraic structure that satisfies the following conditions. These simply express basic rules that can be established to be necessary by considering FSM transition diagrams, and asking the question "what must be true for *B* to be closed in this situation?"

3.1 Regular sets over monoids

Suppose $(B,\otimes)$ is a monoid, *i.e.* a semigroup with an identity element 1. Given any alphabet *A* of the same cardinality as *B*, we set up a 1-1 correspondence $\Lambda: A \to B$. This extends to a monoid homomorphism $\Lambda^*: A^* \to B$ given by $\Lambda(\varepsilon) = 1_B$, and $\Lambda(b_1,\ldots,b_n) = b_1 \otimes \ldots \otimes b_n$. This in turn extends to a function Λ^{**}: LANG$(A) \to \wp B$ given by $\Lambda^{**}(L) = \{ \Lambda^*(\mathbf{a}) \mid \mathbf{a} \in L\}$. If *L* is a regular language over *A*, we'll say that $\Lambda^{**}(L)$ is a regular subset of *B*. The set of all regular subsets of *B* will be written REG(B). It includes $\varnothing = \Lambda^{**}(\varnothing)$ and all finite subsets of *B*.

Notice that $\wp B$ is itself a monoid, with identity element $\{1_B\}$, where we define $U \otimes V = \{u \otimes v \mid u \in U, v \in V\}$, so it is meaningful to discuss regular subsets of $\wp B$. Moreover, REG(B) is also a monoid with these choices of operation and identity.

3.2 Definition (Behavioural framework)

We will say that a triple $B = (B,\otimes,\Sigma)$ is a (*behavioural*) *framework* provided

1. $(B,\otimes)$ is a semigroup, with a 2-sided identity element, 1_B. We call $\otimes$ *concatenation*.
2. Σ: REG(B)$\rightarrow B$ is a function, called (*regular*) *aggregation*, satisfying
    a. $\Sigma\varnothing = 1_B$
    b. $\Sigma\{b\} \equiv b$
    c. $\Sigma b_i \otimes \Sigma b_j \equiv \Sigma\{\ b_i \otimes b_j\ \}$
    d. $\forall$ regular $\{V_i \mid i \in I\} \subseteq$ REG(B) : $\Sigma \cup \{V_i \mid i \in I\} \equiv \Sigma\{\ \Sigma V_i \mid i \in I\}$

If B is a framework, we can define a commutative binary addition $\oplus$ by $u \oplus v = \Sigma\{u,v\}$. ■

Frameworks are ubiquitous in theoretical computer science and mathematics. The distributive laws (2.c) means that frameworks are closely related to algebraic rings, though the concepts are not identical because $\oplus$ need not allow inverses. In particular, if X is any datatype, the monoid $R(X)$ of all relations $X \leftrightarrow X$ is a framework, when we take $\otimes$ to be relational composition, and $\oplus$ to be set-theoretic union. Likewise, if we treat Σ as *inf* and $\otimes$ as *sup*, then every frame (hence, every topology) is a framework, but again the converse is not true, because we don't require $\otimes$ to be commutative.

3.3 Definition (β–machine)

A *behavioural machine* (*β–machine*) is a pair (F,Λ), written F^Λ, where F is a semi-automaton over some alphabet A, and Λ: $A \rightarrow B$ is a function, where B is a behavioural framework.

Given a string **a** recognised by F, the *path behaviour* computed by **a** is the behaviour $\Lambda(\mathbf{a})$, where we define $\Lambda(\varepsilon) = 1_B$ and $\Lambda(a_1 \ldots a_n) = \Lambda(a_1) \otimes \ldots \otimes \Lambda(a_n)$. The *behaviour* of F^Λ is the aggregate of its path behaviours, $|F^\Lambda| = \Sigma\{\ \Lambda(\mathbf{a}) \mid \mathbf{a} \in |F|\ \}$.

3.4 Examples

3.4.1 LANG(A)

The set LANG(A) of languages over A is a framework, where $\otimes$ is standard language concatenation, and Σ is set-theoretic union.

3.4.2 MAZURKIEWICZ TRACE LANGUAGES

(Mazurkiewicz) trace languages were introduced in [Maz77; see also CF69] as a tool for modelling Petri net behaviours. They have since become one of the most popular models for concurrent semantics [Gas90, Arn91, KS92, Sta94, DR95, DG98, DG02].

Each trace language can be viewed as a quotient $A^*/\equiv$, where $\equiv$ is a congruence on A^*. That is, whenever $\mathbf{a} \equiv \mathbf{b}$, we also have $\mathbf{ac} \equiv \mathbf{bc}$ and $\mathbf{ca} \equiv \mathbf{cb}$, for all **c**. This congruence ensures that

concatenation of traces is well-defined, and we can construct languages (subsets of $A^*/\equiv$) just as we could construct LANG(A).

3.4.3 REG(A)

The set REG(A) $\subseteq$ LANG(A) of regular languages over A is a sub-framework of LANG(A). This is not obvious – indeed, our formulation of behavioural frameworks was originally motivated by the related question: *under what circumstances is a union of regular languages again regular?* Obviously, an *arbitrary* union of regular sets need not be regular, because any non-empty language is the union of its (necessarily regular) singleton subsets. The framework concept embraces one answer to this question: *any regular union of regular languages is again regular*. This is what we now prove. We need to do this in any case, to verify that our choice of aggregation operator Σ is well-defined. In general, aggregation is represented by the operator Σ: REG(B) $\rightarrow B$, and since we are taking B = REG(A), our aggregation operator must have signature Σ: REG(REG(A)) $\rightarrow$ REG(A). By hypothesis, we are taking Σ to be set-theoretic union, $\cup$, so we need to verify that whenever we are given a regular set of regular languages, their union is again a regular language.

Theorem. Any regular union of regular languages is regular. Formally, suppose that $\{L_i \mid i \in I\}$ is a regular subset of REG(A), and let $L = \cup\{L_i \mid i \in I\}$. Then $L \in$ REG(A).

Proof. By definition, there is some regular language L_0 over some alphabet A_0, and some labelling function Λ_0: $A_0 \rightarrow$ REG(A) with $\Lambda_0{}^{**}(L_0) = \{L_i \mid i \in I\}$. Because L_0 is regular over A_0, it can be generated by a regular expression e_0 over just finitely many of the symbols in A_0, so without loss of generality we can assume that A_0 is finite. We can also assume that A_0 is non-empty, since otherwise L would be finite and we'd have nothing to prove. Without loss of generality, then, we can write $A_0 = \{a_{01}, \ldots, a_{0n}\}$ for some finite non-zero n. Setting $L_{0i} = \Lambda_0(a_{0i})$, we observe that each L_{0i} is a regular language of A, so we can choose regular expressions $e_1, \ldots, e_n$, all defined over A, where each e_i generates the corresponding L_{0i}. Let e be the expression over A obtained by simultaneously replacing every occurrence of each symbol a_{0i} in e_0 with the corresponding expression e_i. Then e is a regular expression over A. It is now straightforward (though rather tedious) to show that e generates the required union, $L = \cup\{L_i \mid i \in I\} = \cup\Lambda_0{}^{**}(L_0)$. So L is regular. ■

3.4.4 AUTOMATA

As we saw in the introductory preamble, semi-automata are simply β–machines over B = REG(A). Automata are slightly more complex, but essentially straightforward. We won't prove the details here, because the methods are identical to those used in the more difficult case of stream X-machines, below.

3.4.5 X-MACHINES

The X-machine is a computational model introduced by Eilenberg [Eil74], that forms the basis for many subsequent models of computation [Sta90, Sta91, Lay93, LS93, BeH96, BCG+99, Sta01]. At its heart, an X-machine is essentially a semi-automaton whose transitions are labelled by operations of type $X \leftrightarrow X$. Each successful path through the machine corresponds to an operation obtained by composing the relations on the various transitions, and the behaviour of the entire machine is defined to be the union of the path behaviours. Clearly, this is just a special case of the β–machine construction, where we take B to be the relational monoid $R(X)$ of all relations on X, with $\otimes$ equal to relational composition and Σ equal to set-theoretic union.

3.4.6 STREAM X-MACHINES

Stream *X*-machines were introduced by Laycock [Lay93] as a version of the *X*-machine equipped with the stream handling capabilities of an automaton. The SXM captures the idea that a system handles inputs by changing its internal memory states; an SXM is given by a semi-automaton whose transitions are labelled by operations of type $Mem \times In \leftrightarrow Out \times Mem$. If the system memory is currently *mem*, and the input *in* is received, then crossing a transition labelled ϕ causes the machine to change memory state to *new_mem* and to generate some output *out*, where $(out, new\_mem) \in \phi(mem, in)$. Clearly, all automata (hence all semi-automata) can be modelled as SXMs with unchanging memory. In fact, all SXMs can be modelled as standard *X*-machines.

Given an SXM with operations of type $Mem \times In \leftrightarrow Out \times Mem$, we represent it as an *X*-machine with type $X = Out^* \times Mem \times In^*$. Each relational label ϕ: $Mem \times In \leftrightarrow Out \times Mem$ is represented as the associated relation ϕ^*: $X \leftrightarrow X$ given by

$$\phi^*(\ \mathbf{out}, mem, in::\mathbf{in}) = \{\ (\mathbf{out}::out, new\_mem, \mathbf{in}) \mid (out, new\_mem) \in \phi(mem, in)\ \}$$

In this form it's easy to see where the name comes from. The behaviour of an SXM causes input streams to be converted into output streams, with the exact conversions being mediated by an embedded *X*-machine of type *Mem*. Since all *X*-machines are β–machines, the same is true of SXMs and automata. ■

4 A general test method for β–machines

A key motivation behind Laycock's introduction of the SXM was the existence of a general test-generation technique for automata. *Chow's W-method* is based on the idea that states in an automaton can be *characterised* by input-output behaviours.

4.1 Chow's *W*-method

We presuppose that a system is specified as an automaton, *Spec*, and that a second automaton, *Imp*, has been generated as an implementation. We have complete knowledge of *Spec*, and can therefore assume that it is minimal (if not we minimise it). For technical reasons, we need to assume that *Spec* and *Imp* are defined on the same alphabet *A*, and that we can reliably estimate the number of extra states in *Imp*. Our goal is to determine whether or not |*Imp*| exactly matches |*Spec*|, and our lack of knowledge of *Imp*'s internal structure forces us to make this determination by supplying various input streams, and observing the corresponding outputs. The basic strategy of Chow's method is as straightforward as it is elegant. We supply a set of input sequences that take us to each state of the machine in turn, and check, once we get there, that it admits precisely the right set of 'next-state' transitions. In this respect it has much in common with other early test techniques [Moo56, Koh78].

4.1.1 DEFINITION (CHARACTERISATION SET FOR AUTOMATA)

Recall that *S* is the state set of the machine *F*. Given any $W \subseteq In^*$, we can define a function f_W: $S \rightarrow [W \leftrightarrow Out^*]$ by

$$f_W(s)(\mathbf{w}) = \begin{cases} \mathbf{out} & \text{if } s \Rightarrow^{(\mathbf{w},\mathbf{out})} T \\ undefined & \text{if no such path exists} \end{cases}$$

If the function f_W is 1-1 and each of the relations $f_W(s)$ is actually a function $f_W(s)$: $W \rightarrow Out^*$ (so that each input stream **w** generates a single well-defined output stream **out**), then W is a *characterisation set.* ■

Remark. For F to have a characterisation set, it must be minimal. For if F is not minimal, there will be states $s \neq s'$ which cannot be distinguished by their contributions to input-output behaviour, and under such circumstances we must have $f_W(s) = f_W(s')$: so W would fail to be a characterisation set.

Characterisation sets allow us to decide what state the machine must have been in, and can be used to check that test sequences have caused the machine to migrate to a required home state. To construct a test set, we first choose a cover C for F (this requires F to be accessible and deterministic), and then a characterisation set, W (requires F to be minimal). To test an implementation we use C repeatedly to visit each state in turn (this typically requires F to be connected, i.e. to have a *reset* mechanism); then we supply each symbol in A (requires F to be complete), so as to exercise every transition; then we check that the transition took us to the appropriate *next*-state using W.

4.1.2 THEOREM (CHOW)

Suppose that *Spec* and *Imp* are automata over the same alphabet A. Suppose, moreover, that *Spec* is minimal, complete, connected and deterministic (so it has precisely one initial state), and that all states are final. Let C be a cover, and W a characterisation set, for *Spec*. Suppose *Imp* has no more than n more states than *Spec*. Then *Spec* and *Imp* are equivalent if and only if they generate the same output string for every input string in $CA^{(n+1)}W$. ■

Remarks. (1) In its original form Chow's theorem takes the test set to be of the form $TA^{(n)}W$, where T is a 'transition cover' for F. In order to keep things simple, our statement of the theorem uses the fact, for complete machines, that $CA^{(1)}$ is a transition cover whenever C is a state cover, so that a suitable test set is $CA^{(1)}A^{(n)}W = CA^{(n+1)}W$. (2) Recalling our method for machines with non-final states, it is clear that Chow's method can be extended to infinitely many more specifications than currently included.

4.2 SXM Testing

The strong similarity between SXMs and automata was the original motivation for their introduction; Laycock [Lay93] recognised that Chow's method could be extended to cover systems specified by SXM. The technique was largely perfected in [Ipa95, IH97, HI98] and has been extended more recently to include non-deterministic specifications [HH01a, HH01b]. It has been applied both theoretically and in practical system testing [FHI+95, BH01, Van01], and support tools are becoming available [EK00, KEK00a, KEK00b].

As with Chow's method, we assume the existence of a minimal, complete, connected and deterministic SXM specification *Spec*, and an SXM implementation *Imp*, and need to decide whether or not they have the same behaviour. The basic strategy involves constraining *Spec* so that the equivalence of *Spec* and *Imp* follows from that of their underlying automata.

Recall our basic requirement that *Spec* and *Imp* should be defined over the same alphabet. In the present context, this means that they should use the same transition relations, or equivalently, that they use the same labelling, Λ.

In Chow's method, characterisation is used to identify machine states, and relies on the fact that individual letters can easily be distinguished (the output streams $\mathbf{out_1}$ and $\mathbf{out_2}$ are distinct if and only if they differ in one or more letters, so we only need to be able to distinguish the

latter).This principle no longer applies for SXMs, because it is impossible to guarantee *a priori* that distinct label relations can be distinguished. For example, suppose two relations differ only at one particular (*mem*, *in*) pair. Unless we know *a priori* that we need to test the relations with this particular (*mem*, *in*) combination, we could wrongly conclude that the correct relation is implemented when in fact it is not. This is a particular problem because *Mem* may well be infinite and even uncountable, so we cannot exhaustively test all (*mem*, *in*) pairs. Consequently, we strengthen characterisation by requiring *output distinguishability*; there should be *a priori known* (*mem*, *in*) pairs that can be used to distinguish between the relations used to label the transitions of the machine.

The concept of *cover* now needs to be strengthened as well, because it is not enough to ensure that we reach every state in turn. When we reach a given state *s*, we need to be certain that *mem* has taken on a value suitable for characterisation to be possible – we have to arrive at *s* with *mem* appropriately set for firing transitions with the key (*mem*,*in*) input pairs. This is typically established by imposing the (somewhat stronger) requirement that for each *mem*, there is some *a priori known* input *in* for which (*mem*, *in*) can be processed and at the same time contribute to the characterisation process. This can be regarded as the SXM version of completeness, though it is not quite the published Holcombe/Ipate "test completeness" property [HI98, p. 181], which omits the contribution-to-characterisation requirement.

Finally, of course, it is not enough just to know that *Imp* is giving the correct responses, because this could occur by accident. If the implementation is written in a programming language whose compiler is faulty, we could well be generating the correct responses for each of the key (*mem*, *in*) test inputs, and yet still not have a correct system. We therefore impose the requirement that the relation labels themselves must be *a priori* correct.

4.2.1 THEOREM (HI98, P.185)

Suppose that *Spec* and *Imp* start with the same initial memory value, use the same set of relation labels Φ, and that Φ is output distinguishable and complete [in our sense, above]. Assume that the associated automaton of *Spec* is minimal, complete, connected and deterministic (so it has precisely one initial state), and that all states are final. Suppose also that the associated automaton of *Imp* has no more than *n* more states than that of *Spec*. Then *TS* is a test set for |*Spec*| where *TS* is Chow's test set. ■

It is normally stated that the correctness of labels can be determined recursively, by applying this same technique to the labels themselves (see *e.g.* HI98, chapter 8). This is only partially true, however, because we have assumed extra properties for these relations as part of the price for using them during testing. It is not enough simply to prove that the label relations are correct in their own terms, because they *also* have to satisfy the consequences of our completeness and output distinguishability criteria. This means we have to prove properties that hold for all values of *Mem*, and this may not be a tractable, or even a possible, problem. For example, the function *test*: $\mathbf{N} \rightarrow$ BOOL given by the program {*while(n>1){ if (even(n)) n=n/2; else n = 3*n+1;} return true;*} is clearly computable, and we can even prove that it correctly calculates a particular specification of interest to number theorists, Consequently, we can (by traditional arguments) use *test* as a valid label in an SXM, because it is correct in its own terms. But this is not valid, because no-one currently knows whether *test* is a partial or a total function – we can prove its correctness in its own terms, but we may not necessarily be able to prove the *extra* requirement, totality, entailed by our completeness criteria.

4.3 General behavioural testing

It is clear that the SXM test method relies on properties that can be expressed entirely in framework-theoretic language. It is not surprising, therefore, that the method can be extended at a stroke to include all behavioural frameworks.

Henceforth, suppose that $B = (B, \otimes, \Sigma)$ is a framework, and that F^{Λ} is a β–machine with alphabet A. Write $\Phi = \Lambda(A) \subseteq B$. We call the members of Φ *behavioural labels*. Recall also that theoretical caution requires us to regard test sets as downward closed; this is reflected in our statement of Theorem 4.3.3 below.

The first step is to define what we mean by a characterisation set for a β–machine, because we can no longer rely on the output language *Out** for characterisation purposes (it isn't part of the general framework definition).

4.3.1 DEFINITION (CHARACTERISATION SET FOR SEMI-AUTOMATON)

Given any $W \subseteq A^*$, we can define a function f_W: $S \to W \to \text{BOOL}_\perp$ by

$$f_W(s)(\mathbf{a}) = \begin{cases} \text{T} & \text{if } s \Rightarrow^{\mathbf{a}} T \\ \text{F} & \text{if no such path exists} \end{cases}$$

If the function f_W is 1-1, we call the language W a *characterisation set* for F. If, in addition, for each s there is at least one $\mathbf{a} \in W$ for which $f_W(s)(\mathbf{a}) = \text{T}$, we call W a *positive* characterisation set. If F, G are semi-automata over A, we say that two states $s \in F$ and $t \in G$ are *W-equivalent* if $f_W(s) = f_W(t)$, where $f_W(s)$ is evaluated in F and $f_W(t)$ in G. ■

It is worth recalling that characterisation sets can only be constructed for minimal machines, because non-minimal machines contain indistinguishable states. We can generally construct a positive characterisation set piecewise. That is, we first find languages that distinguish s_1 from s_2, for each pair (s_1,s_2), and then take the union of all of these component languages. In general this is less efficient than constructing a global characterisation set, but may be simpler in practice.

4.3.2 DEFINITION (DISTINGUISHABLE BEHAVIOUR LABELS)

Let ∂: $\Phi \to [B \to \text{BOOL}]$ be the function which maps each $\phi \in \Phi$ to the predicate

$$\partial(\phi)(b) \equiv (b = \phi)$$

We call ∂ the *distinguishing function* for Φ. If $\partial(\phi)$ is decidable for all $\phi \in \Phi$, we say that Φ is *distinguishable*. ■

In general, this is the best we can do, because the actual proof that ∂ generates decidable predicates will depend on the exact nature of B. For SXMs, for example, we avoid the requirement that $\partial(\phi)$ be decidable by imposing the stronger requirement of output-distinguishability instead.

4.3.3 THEOREM (COMPLETE BEHAVIOURAL TEST SETS)

Suppose that $Spec^{\Lambda}$ and Imp^{Λ} are β–machines over the same framework with the same Φ, and that Φ is distinguishable. Suppose that *Spec* is minimal, complete, connected and deterministic (so it has precisely one initial state), and that all states are final. Let C be a cover and W a

positive characterisation set for *Spec*. Suppose *Imp* has no more than *n* more states than *Spec*. Suppose also that the initial termination behaviours of *Spec* and *Imp* are identical (i.e. the unique initial states of the two machines are *W*-equivalent). Then $Spec^{\Lambda}$ and Imp^{Λ} have the same behaviour if and only if they generate the same behaviour for every input string in $(CA^{(n+1)}W)\downarrow$.

Proof. We will show that Chow's theorem can be applied to the F/T automaton representing *Spec*. To this end, we have to choose some $\mathbf{a} \in CA^{(n+1)}W$ and consider the stream translation function f^* defined as follows, where $f = f_{A^*}(\iota)$ and ι is the initial state of *Spec*.

$$f^*(a_1, \ldots, a_n) \mapsto (f(a_1),\ f(a_1a_2), \ldots, f(a_1, \ldots, a_n))$$

In other words, the sequence on the right simply gives the acceptance/rejection history of the string, as it evolves one symbol at a time. This is the automata-style behaviour of *Spec*, and we need to show that it is the same as the automata style behaviour of *Imp*. It will then follow immediately that $|Spec| = |Imp|$, and that, therefore, $|Spec^{\Lambda}| = |Imp^{\Lambda}|$.

By hypothesis, $Spec^{\Lambda}$ and Imp^{Λ} behave identically on all prefixes of **a**. In particular, $|Spec^{\Lambda}|$ is defined for the prefix **u** if and only if $|Imp^{\Lambda}|$ is also defined for **u**, and this occurs if and only if $\mathbf{u} \in |Imp|$. Consequently, the corresponding stream translation function for *Imp*

$$g^*(a_1, \ldots, a_n) \mapsto (g(a_1),\ g(a_1a_2), \ldots, g(a_1, \ldots, a_n))$$

satisfies $f^*(\mathbf{u}) = \mathrm{T} \Leftrightarrow g^*(\mathbf{u}) = \mathrm{T}$ as required, and $f^*(\mathbf{u}) = g^*(\mathbf{u})$. ■

5 Summary

We have provided two extensions to state-machine testing theory. The first allows us to extend Chow's *W*-method and the SXM testing method to include specifications given by machines with non-final states, by first encoding semi-automata as automata with output alphabet BOOL. The second part of the paper demonstrates a general extension to these test methods, that applies to any system that can be specified by a behavioural framework. Although a novel concept, the frameworks in question occur ubiquitously in computer science, with examples occurring in both sequential and concurrent machine theory. We show that Eilenberg's *X*-machine concept can be extended to give *behavioural machines*, which are essentially semi-automata whose transitions are labelled by elements of a behavioural framework. The SXM test method extends automatically to behavioural machines, provided we recognise the different natures of semi-automaton and automaton behaviours. Accordingly, we need to use the downward closure of Chow's test set to generate the test-set for a semi-automaton-based β–machine.

There is clear scope for future work in this field. In particular, we are studying the feasibility of modelling object oriented systems using β–machines, by incorporating both the sequential and concurrent representational capabilities into a single framework. Given that tools are becoming available for SXM testing, we are hopeful that tools can likewise be developed for β–machine testing, though any such tools will require considerable built-in user support.

6 References

[Arn91] Arnold A. (1991) 'An extension of the notion of traces and asynchronous automata.' *Theoretical Informatics and Applications* **25**, pp. 355-393.

[BCG+99] Balanescu, T., Cowling, T., Georgescu, H., Gheorghe, M., Holcombe, M. and C. Vertan (1999) 'Communicating stream X-Machine systems are no more than X-machines.' *J.U.C.S.* **5**(9), pp. 492-507.

[BeH96] Bell, A. and M. Holcombe (1996) “Computational models of cellular processing” in R. Cuthbertson, M. Holcombe and R. Paton, eds., *Computation in Cellular and Molecular Biological Systems*. Singapore: World Scientific.

[BH01] Bogdanov, K. and M. Holcombe (2001) ‘Statechart testing method for aircraft control systems.’ *Software Testing, Verification and Reliability* **11**, pp. 39-54.

[CF69] Cartier, P. and D. Foata (1969) Problèmes combinatoires de commutation et rearrangements. (Lecture Notes in Mathematics 85), London & Berlin: Springer-Verlag.

[Cho78] Chow, T. (1978) ‘Testing software design modelled by finite state machines.’ *IEEE Transactions on Software Engineering* **SE-4**(3), pp. 178-187.

[DG98] Diekert, V. and P. Gastin (1998) ‘Approximating traces.’ *Acta Informatica* 35, pp. 567-593.

[DG02] Diekert, V. and P. Gastin (2002) “Safety and Liveness Properties for Real Traces and a Direct Translation from LTL to Monoids” in W. Brauer et al., eds., *Formal and Natural Computing* (*Lecture Notes in Computer Science* 2300), London & Berlin: Springer-Verlag; pp. 26-38.

[DR95] Diekert, V. and G. Rozenberg, eds. (1995) *The Book of Traces*. Singapore: World Scientific.

[Eil74] Eilenberg, S. (1974). *Automata, Languages and Machines, Vol. A*. Academic Press.

[EK00] Eleftherakis, G. and P. Kefalas (2000) *Model Checking X-Machines: Towards integrated formal development of safety critical systems*. Technical Report, Dept of Computer Science, CITY Liberal Studies, Thessaloniki, Greece.

[FHI+95] Fairtlough, M., Holcombe, M., Ipate, F., Jordan, C., Laycock, G. and Z. Duan (1995) ‘Using an X-machine to model a Video Cassette Recorder.’ *Current issues in Electronic Modelling*, **3**, pp. 141-161.

[Gas90] Gastin, P. (1990) ‘Un modèle asynchrone pour les systèmes distributes.’ *Theoretical Computer Science* **74**, pp. 121-162.

[HH01a] Hierons, R. and M. Harman (2001) ‘Testing conformance to a quasi-nondeterministic stream X-machine.’ *Formal Aspects of Computing (Special Issue on X-machines)*.

[HH01b] Hierons, R. and M. Harman (2001) ‘Testing conformance of a deterministic implementation against non-deterministic stream X-machine.’ Working Paper, Dept of Computer Science, Brunel University, UK.

[HI98] Holcombe, M. and F. Ipate (1998) *Correct Systems: Building a Business Process Solution*. London & Berlin: Springer-Verlag.

[Hol89] Holcombe, M. (1989) ‘X-machines as a basis for dynamic system specification.’ *Software Engineering Journal* **3**(2), pp. 69-76.

[IH97] Ipate, F. and M. Holcombe (1997) ‘An integration testing method that is proven to find all faults.’ *International Journal of Computer Mathematics* **68**, pp. 159-178.

[Ipa95] Ipate, F. (1995) *Theory of X-machines with Applications in Specification and Testing*. PhD Thesis, Dept of Computer Science, Sheffield University.

[KEK00a] Kefelas, P., Eleftherakis, G. and E. Kehris (2000) *Communicating X-Machines: A Practical Approach for Modular Specification of Large Systems*. Technical Report CS-09/00, Department of Computer Science, CITY Liberal Studies, Thessaloniki, Greece.

[KEK00b] Kehris, E., Eleftherakis, G. and P. Kefalas (2000) “Using X-Machines to Model and Test Discrete Event Simulation Programs” in N. Mastorakis, ed., *Systems and Control: Theory and Applications*. Singapore and America: World Scientific and Engineering Society Press; pp. 163-168.

[Koh78] Kohavi, Z. (1978) *Switching and finite automata theory*. McGraw-Hill.

[KS92] Kwiatkowska, M. and M. Stannett (1992) “On Transfinite Traces” in V. Diekert and E. Ebinger, eds., *ASMICS Workshop on Infinite Traces*, Bericht 4/92, Universität Stuttgart Fakultät Informatik; pp. 123-157.

[Lay93] Laycock, G. (1993) *The Theory and Practice of Specification-Based Software Testing*. PhD Thesis, Dept of Computer Science, Sheffield University, UK.

[LP84] Lidl, R. and G. Pilz (1984) *Applied Abstract Algebra*. London & Berlin: Springer-Verlag.

[LS93] Laycock, G. and M. Stannett (1993) *X-machine Workshop*. Technical Report, Dept of Computer Science, Sheffield University, UK.

[Maz77] Mazurkiewicz, A (1977) *Concurrent program schemes and their interpretations*. Technical Report DAIMI Pβ−78, Aarhus University, Denmark.

[Moo56] Moore, E. F. (1956) ‘Gedanken-experiments on sequential machines.’ *Automata Studies*, (*Annals of Mathematics Studies, no 34*). Princeton University Press; pp. 129-153.

[Sta90] Stannett, M. (1990) ‘X-machines and the Halting Problem: Building a super-Turing machine.’ *Formal Aspects of Computing* **2**, pp. 331-341.

[Sta91] Stannett, M. (1991) *An Introduction to post-Newtonian and non-Turing Computation*. Technical Report CS-91-02, Dept of Computer Science, Sheffield University, UK. Available online from http://www.dcs.shef.ac.uk/research/resmems/.

[Sta94] Stannett, M. (1994) ‘Infinite Concurrent Systems – I. The Relationship between Metric and Order Convergence.’ *Formal Aspects of Computing* **6**, pp. 696-715.

[Sta01] Stannett, M. (2001) *Computation over arbitrary models of time.* Technical Report CS-01-08, Dept of Computer Science, Sheffield University, UK. Available online from http://www.dcs.shef.ac.uk/research/resmems/.

[Van01] Vanak, S. K. (2001) *Complete Functional Testing of Hardware Designs*. Preliminary PhD Report, Dept of Computer Science, Sheffield University, UK.

Formal Basis for Testing with Joint-Action Specifications*

Timo Aaltonen and Joni Helin
Tampere University of Technology, Institute of Software Systems
P.O.BOX 553, FIN-33101 Tampere, Finland
Tel: +358-3-31153951, fax: +358-3-31152913
email: {timo.aaltonen, joni.helin}@tut.fi

July 3, 2002

Abstract

A new method for formal testing of reactive and distributed systems is presented. The method is based on having a joint-action specification of the implementation under test. A testing hypothesis is made to allow assigning formal meanings to correct and incorrect implementations. Utilizing these we formulate meaning of finding errors in implementations in a strict way.

1 Introduction

Traditionally testing and formal methods have had little to do in common. However, recently it has been realized that these two need not be exclusive but they can complement each other. Rigorous specifications give an excellent basis for testing: notions like correct and erroneous implementations can be assigned an exact meaning and descriptions of test cases become more precise.

One difficult field for testing are reactive and distributed systems. Reactive systems obtain stimuli from the environment and react to them. The behaviour of a distributed system is not very predictable. Little changes in timings of their complex interactions with the environment and between components make the system behave in non-deterministic manners. Therefore testing them thoroughly is extremely difficult. Formal specifications allow us to deal with this non-determinism in an exact way.

In this paper we attack problems of testing reactive and distributed systems by utilizing joint-action specifications which are introduced in Section 2. Systems are first modelled with them and then they are implemented. We make

*This work was supported by Academi of Finland under project 510005.

a testing hypothesis which claims that there exists a joint-action model for every implementation. If an implementation is correct then this model can be achieved from the specification as a legal refinement. Testing is searching for a counterexample to this refinement relation. These aspects are discussed in Section 3. In Section 4 an example of utilizing the proposed testing method is given. Finally Section 5 concludes the paper.

2 Joint-Action Specifications

Joint actions were introduced by Ralph Back and Reino Kurki-Suonio in [4, 5]. The intention with them was to describe the *behaviour*s of reactive and distributed systems at a high level of abstraction. By abstracting the details of communication, a joint action models *what* several parties of the system do together, not *how* the desired behaviour is achieved.

Basic building blocks of our joint-action specifications are class and action declarations. Formally a class is a set of similar objects sharing the same structure. Classes can be inherited in an object-oriented fashion. Then an inherited class forms a subset of the base class. Besides class definitions, record types can be declared. Unlike objects of classes, *values* of record types have no identity.

The state of an object means evaluation of its attributes. The states of objects can be changed only by executing multi-object joint actions. A joint action consists of a name, a set of *roles* (in which the objects can *participate*), a boolean-valued *guard* (which must evaluate to *true* for the participating object combination) and a body (which is a parallel assignment clause assigning new values to the attributes of the participating objects). When such a combination of objects exists that the guard of an action is satisfied, an action is said to be *enabled* (for the satisfying object combination). Selection of the next action to be executed is non-deterministic among the enabled ones. Actions are atomic units of execution which are executed in an *interleaving* manner. This is an abstraction of parallel executions of operations.

States of all objects constitute the global state of the system. A new global state is obtained form the current one by executing an action which is enabled in the current state. A sequence of global states starting from an initial state s_0 is called a *behaviour*: $\sigma =< s_0, s_1, s_2, ... >$ where s_0 is an initial state of the system and each state s_{k+1} is obtained from state s_k by executing an action enabled in state s_k. The method is not sensitive to *stuttering* in behaviours. In other words a behaviour is considered the same even if its states are arbitrarily repeated. A joint-action specification induces potentially an infinite set of behaviours whose initial states are legal initial states of the specification and new states are obtained by executing the actions of the specification.

As an example we give a specification of counters. Objects belonging to class `Counter` hold the value they count in attribute `val`, which can be incremented by executing action `inc`. The action can be executed for objects having value less than two in `val`; the body of the action increments `val` by one. It is assumed that all counters are initialized to zero:

```
specification Cntrs = {
  class Counter = {val: integer}
  action inc(c: Counter): c.val < 2 ->
    c.val′ = c.val + 1;
  }
```

From the infinite set of behaviours the specification induces we show two. In the first one just one counter has been instantiated, the value of attribute `val` is given in parenthesis: $< (0), (1), (2), (2), \cdots >$. The second behaviour consists of two counters: $< \binom{0}{0}, \binom{1}{0}, \binom{2}{0}, \binom{2}{1}, \binom{2}{2}, \binom{2}{2}, \cdots >$.

Formal properties of behaviours are often divided into two categories: *safety* and *liveness* properties. The former are such their violation can be detected in a finite prefix of a behaviour. In terms of temporal logic[10, 13] an example of safety property is $\Box p$, which states that proposition p holds in every state. For example, $\Box(\forall c : Counter :: c.val < 3)$ holds for specification `Cntrs`, if counters are initialized to zero. On the other hand liveness properties are those whose violence cannot be detected in a finite prefix. An example of such is $\Diamond q$ which states that eventually q will hold. In this paper we concentrate on safety properties.

2.1 Refinement

Joint-action specifications are refined towards implementations by superimposing new layers onto an old specification. A superposition step maps the old specification to a new one, which is a refinement of the old one. Our variant of superposition allows adding new attributes to existing classes, introducing new classes, strengthening the guards of old actions, adding new assignments to old actions, introducing new actions, but the new assignments are allowed only to newly introduced attributes.

In terms defined in [8] our variant of superposition is *regulative*. I.e., the safety properties of the specification being refined are preserved by construction, but liveness properties can be violated. This means that if we have an abstract specification s and its refinement s' then all behaviours induced by s' (denoted $beh(s')$) are also behaviours of s with respect to variables of s: $beh(s') \subseteq_{vars(s)} beh(s)$.

As an example, specification `Cntrs` can be refined by adding `running` bit to each counter and introducing actions to set and reset the bit. The guard of action `inc` is strengthened with a conjunct stating that `running` must be *true* for the participating counter; the body of the action remains unchanged:

```
specification Ena refines Cntrs = {
  Counter = Counter + {running: boolean}
  action enable(c: Counter): not c.running -> c.running′ = true;
  action disable(c: Counter): c.running -> c.running′ = false;
  action inc(c: Counter) refines Cntrs.inc(c): c.running -> ...;
  }
```

In a refinement the values of abstract variables can be represented by several variables distributed in the system. In these cases quantified expressions called *shadow assertions* state formally the relationship between the abstract and the

concrete variables. To ensure that shadow assertions are not violated they must be formally verified. When a variable has become abstract then it is not needed in an implementation.

Several joint-action specifications can be composed into one compound specification. Applying refinement and composition lead to abstraction hierarchies, which are directed acyclic graphs.

2.2 Abstraction Function

Remember that our refinement mechanism allows only adding new variables to the specification and strengthening guards of existing actions and making such augmentations to the bodies of existing action that assign only to newly introduced variables. Therefore, if specification s' is a refinement of s then each behaviour induced by s' has an image in behaviours induced by s in terms of s.

To be able to deal with behaviours of one specification in an abstraction hierarchy at the level of another, we define *abstraction function*: $\mathbf{af}^{s}_{s'} : B \mapsto B$, where s and s' are specifications and B is the universe of behaviours. The function is defined if $vars(s) \subseteq vars(s')$, where $vars(s)$ are the variables (attributes of objects) of specification s. The function maps behaviours induced by specification s' (more concrete) to behaviours of s (more abstract) by filtering variables $vars(s') \cap vars(s)$ from the behaviour and by computing the values of shadow variables of s.

For example, behaviours induced by `Ena` can be mapped to behaviours of `Cntrs` simply by filtering the values of attribute `running` from the behaviour. More formally

$$\mathbf{af}^{Cntrs}_{Ena} \equiv \{(b_{Ena}, b_{Cntrs}) \mid (\forall i \in \mathbb{N} : (\forall c \in Counter : c.val(b_{Ena}(i)) = c.val(b_{Cntrs}(i))))\}$$

where $\mathbb{N}$ is the set of natural numbers and $c.val(b(i))$ is the value of attribute val in the ith state of behaviour b.

3 Testing with Joint-Action Specification

In this paper implementations are created according to the most concrete specification (denoted c) in the specification hierarchy. The objective is to produce an implementation that is a legal *refinement* of c. Our refinement mechanism is superposition, thus the implementation should be obtainable by some superposition step on c. Then the implementation can only restrict (not liberate) the behaviour induced by c. In our approach *conformance* means that an implementation exhibits only such behaviours that are induced by the specification.

3.1 Testing Hypothesis

A difficulty in this approach is that an implementation is not a formal entity but lies in the physical reality, consisting of a compiled program written in some

programming language, pieces of hardware in which the program slices are run, an environment with which the actual system interacts etc. No concepts exist on which the described correctness can be formally based on; we are not able to validate whether the implementation actually is a refinement of its specification. This dilemma is solved by making a *testing hypothesis*, which states that every implementation under test (IUT) has such a formal *model* (m_{IUT}) that if it and IUT are both put in a black box which transmits behaviours only at the abstraction level fixed by m_{IUT} then we cannot distinguish IUT and m_{IUT}. Model m_{IUT} is a joint-action specification. More formally the hypothesis states that

$$\forall\ \mathrm{IUT} \in Imps\ \exists\ m_{\mathrm{IUT}} \in Specs : \mathbf{af'}_{\mathrm{IUT}}^{m_{\mathrm{IUT}}}(beh'(\mathrm{IUT})) = beh(m_{\mathrm{IUT}}) \qquad (1)$$

where $Imps$ is the universe of implementations, $Specs$ is a universe of joint action specifications, $beh' : Imps \mapsto 2^{Behaviours'}$ maps an implementation to the set of real world behaviours it induces ($Behaviours'$ is the universe of real world behaviours), $\mathbf{af'}_{\mathrm{IUT}}^{m_{\mathrm{IUT}}} : Behaviours'\ of\ \mathrm{IUT} \mapsto Behaviours\ of\ m_{\mathrm{IUT}}$ projects the real world behaviours of IUT to the behaviours of model m_{IUT} and function $beh : Specs \mapsto 2^{Behaviours}$ maps a joint action specification to the set of behaviours it induces. $Behaviours$ is the universe of behaviours induced by joint-action specifications.

Believing that m_{IUT} exists is a leap of faith which cannot be formally justified. However, it is easy to believe that all real systems can be modelled with joint action specifications, because their expressive power is reasonably high. For example, a specification which contains images of all the variables of IUT and an action for each atomically executed piece of software works as the required model.

The testing hypothesis does not require m_{IUT} to be actually created, it just assumes that such a model exists. For testing purposes we only need to be able to produce such behaviours that m_{IUT} would produce. To be correct, model m_{IUT} should be a refinement of c. Therefore, behaviours of m_{IUT} can be mapped to the behaviours of c by applying abstraction function $\mathbf{af}_{m_{\mathrm{IUT}}}^{c}$.

3.2 Definitions for Correct and Incorrect Implementation

Correctness of IUT can now be defined formally:

$$\mathrm{IUT}\ is\ correct\ \Leftrightarrow_{def}\ (\forall b \in beh(m_{\mathrm{IUT}}) : \mathbf{af}_{m_{\mathrm{IUT}}}^{c}(b) \in beh(c)) \qquad (2)$$

In an ideal case we could produce a perfect model m_{IUT} and formally verify that it is a legal refinement of c. Unfortunately, for reasonably sized systems producing model m_{IUT} is not an option and, moreover, utilizing it would not be testing but verification. Instead, we investigate a subset of behaviours induced by m_{IUT} and try to catch errors with them. An attempt is to produce behaviour b_{ce} to satisfy the right hand side of Equation (3) below, which is achieved by negating both sides of Equation (2):

$$\mathrm{IUT}\ is\ not\ correct \Leftrightarrow (\exists\ b_{ce} \in\ beh(m_{\mathrm{IUT}}) : \mathbf{af}_{m_{\mathrm{IUT}}}^{c}(b_{ce}) \notin beh(c)) \qquad (3)$$

Behaviour b_{ce} is a counterexample for the subset relation which should hold for behaviours $beh(m_{\text{IUT}}) \subseteq_{vars(c)} beh(c)$.

3.3 Setting

Figure 1 illustrates the setting for testing. Implementation IUT in the bottom left-hand corner is created based on specification c in the top left-hand corner. According to the testing hypothesis m_{IUT} exists and IUT is correct if and only if m_{IUT} is a legal refinement of c. Each specification induces a set of behaviours depicted as ellipses. Arrows from $beh'(\text{IUT})$ to $beh(m_{\text{IUT}})$ model function $\mathbf{af'}_{\text{IUT}}^{m_{\text{IUT}}}$ which projects real world behaviours to behaviours of m_{IUT}, and arrows from $beh(m_{\text{IUT}})$ to $beh(c)$ model mapping $\mathbf{af}_{m_{\text{IUT}}}^{c}$. Our refinement mechanism enforces the set of behaviour images of the refined specification to be in a subset relation with the set of behaviours of the specification being refined. Testing is an attempt to produce a behaviour b_{ce} which is a counterexample to this subset relation: $b_{ce} \in beh(m_{\text{IUT}}) \wedge \mathbf{af}_{m_{\text{IUT}}}^{c}(b_{ce}) \notin beh(c)$. Behaviour b_5^m is such a counterexample in the figure.

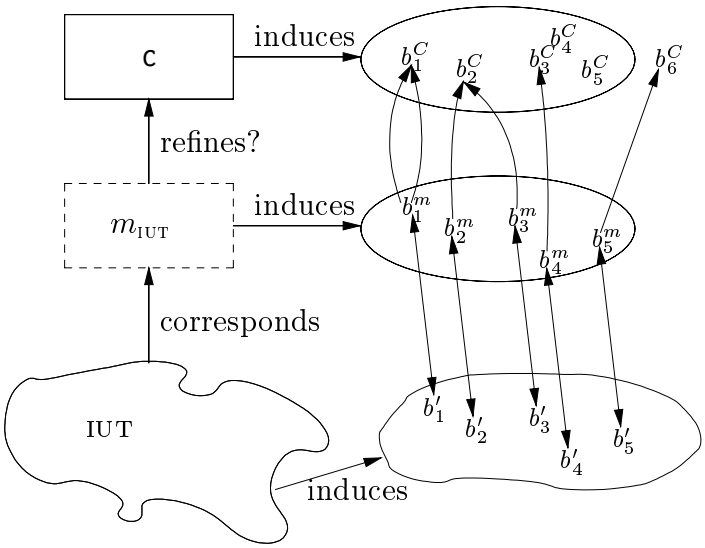


Figure 1: The setting for testing.

3.4 Observation Objectives

So far we have discussed what are correct and incorrect implementations and how errors are detected *if* some behaviours of m_{IUT} are observed. If IUT does nothing then it definitely does not violate the safety properties of specification c. Therefore, a set of behaviours must be enforced. Similarly to [16] we define some

set of behaviours that we whish to observe during testing. These *observation objectives* are given in terms of c.

An observation objective is a set of behaviours that intersects the set induced by c. Objective o is said to be *satisfied* if at least one behaviour $b_o \in o$ is observed. An objective describes behaviours we whish to observe, but it is not directly related to the correctness of IUT. An example objective for specification `Ena` given Section 2 could be the set of behaviours where the value of some counter has first been one and later it has been incremented; more formally objective $o = \{b \mid P(b)\}$ where
$P(b) = \exists i_1, i_2 \in \mathbb{N} \mid i_1 < i_2 : (\exists c \in Counter : c.val(b(i_1)) = 1 \wedge c.val(b(i_2)) > 1)$
where $c.val(b(i))$ is the value of attribute `val` in ith state of behaviour b.

The Venn diagram in Figure 2 depicts different sets of behaviours. The large square is an infinite universe of behaviours. The horizontal large ellipse in the middle models the behaviours induced by specification c (these are legal behaviours); the horizontal small ellipse is an observation objective (these are the behaviours we wish to observe) and the vertical large ellipse models the behaviours $\mathbf{af}^{c}_{m_{\text{IUT}}}(beh(m_{\text{IUT}}))$ (these are the behaviours IUT induces). All the sets are infinite in the general case.

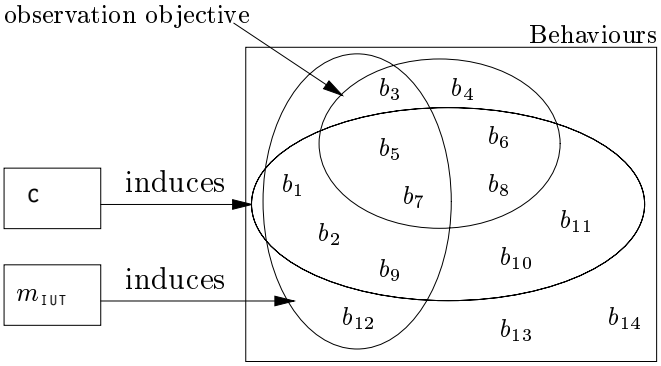


Figure 2: Venn diagram of behaviours.

4 Example: Distributed List

We have applied the proposed testing method for testing an implementation of DISTRIBUTED LIST[12]. Our version of the protocol specification consist of two layers `atomic_list` and `messages`.

4.1 Specification `atomic_list`

DISTRIBUTED LIST is a protocol for maintaining a circular, singly-linked list of cells. The protocol is an abstraction of a part of a multiprocessor cache coherence protocol for linked list of processors sharing a cache line. A more

comprehensive description of the protocol can be found in [12]. In the list there is a special cell, called the *head cell*, which coordinates the addition of cells.

In abstract specification `atomic_list` of the protocol `cell`s can join and leave the list in synchronous actions. First a base class `Cell` with two derived classes `HeadCell` and `NormalCell` are given. Objects of class `Cell` have attribute `next_a` (`a` for abstract), which is a reference to the next cell in the list or a self reference if the `cell` is not in the list. In the initial state there are no cells in the list. Actions `atomicAdd` and `atomicDelete` model joining and leaving the list, respectively.

Action `atomicAdd` has two roles: `nc`, in which the `cell` willing to join the list participates, and `h`, in which the head cell participates. The guard of the action requires that the object participating in role `h` is not yet in the list. In the body of the action reference `next_a` of the head cell is assigned the reference to `nc` and `next_a` of `nc` get the previous value of `h.next_a`:

```
action atomicAdd(nc: NormalCell; h: HeadCell) is
when nc.next_a = nc do
  nc.next_a' = h.next_a ∧ h.next_a' = nc;
end;
```

Detailed description of `atomicDelete` is omitted for brevity.

4.2 Specification `messages`

Synchronous specification `atomic_list` is refined to specification `messages`, which is an asynchronous implementation of the atomic specification. In the refined version a cell not in the list may request to be added to the list and successively to be removed from the list. Communication is message-based over a reliable but not order-preserving medium. Every cell has a specification variable `next_c` (`c` for concrete), which holds the identity of the successor cell, or its own identity if the cell is not on the list.

Cells other than the head cell can perform two types of transactions, *add* and *delete*. Addition consists of sending a request message to the head cell, which results in the cell being added to the list in front of the *next* cell of head cell and a reply containing the identity of the new *next* cell of the requesting cell.

To delete a cell from the list, knowledge of the predecessor cell is required, so a message called `pred` carrying the identity of a cell's predecessor is circulated. A cell initiates deletion by sending a message to its predecessor and waits for acknowledgment. Handling of the deletion message depends on intricate details about the scenario, but basically the actual predecessor of the requesting cell updates its *next_c* variable and acknowledges.

The abstract atomic specification `atomic_list` is superimposed by layer `messages`, in which abstract atomic actions are implemented by more concrete ones, and abstract variable `next_a` is distributed to several concrete variables. The transmission medium is represented with singleton class `Net` holding a set of `Messages`. Class `Cell` is extended with a state machine for message-based implementation of the protocol and variable `next_c`, which together with fields in

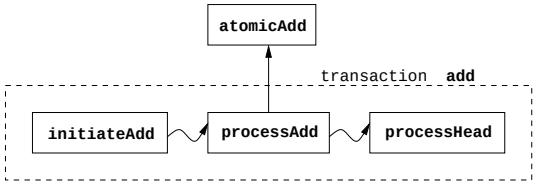


Figure 3: Actions of transaction add.

transmitted messages implement the abstract specification variable next_a. A set of shadow assertions state their respective relationships.

Distributed implementation of the add transaction consists of three actions depicted in Figure 3. The transaction begins when a cell is in normal operating state and wishes to join the list. Action initiateAdd results in an AddMessage sent to the head cell and the cell entering state waiting for a HeadMessage in reply. Set manipulations of Net correspond to message send and reception:

```
action initiateAdd(nc: NormalCell; net: Net) is
when nc.state'normal ∧ nc.next_c = nc do
  net.mess′ = net.mess + {AddMessage(src′ = nc, dst′ = nc.theHeadCell)} ∧
  nc.state′ = w_head();
end;
```

Action processAdd is refined from atomicAdd and locally processed by the head cell upon delivery of the AddMessage. The action guard ties the sender of the message to the *shadow role*[1] nc. Action body consists of sending a reply message carrying the new value for the concrete variable next_c of the requesting cell. The concrete variable in the head cell is updated correspondingly with the abstract variable. Three dots in the guard refer to the guard of the action being refined, and in the body they refer to the original body:

```
refined processAdd(nc: NormalCell; hc: HeadCell; net: Net; am: AddMessage)
of atomicAdd(nc, hc) is
when ... am ∈ net.mess ∧ am.dst = hc ∧ am.src = nc do
  ...
  net.mess′ = net.mess - {am} + {HeadMessage(src′ = hc,
                                             dst′ = am.src,
                                             new′ = hc.next_c)} ∧
  hc.next_c′ = am.src;
end;
```

Transaction add is finished when the initiating cell executes processHead action. The cell updates its concrete variable next_c with the value received in the HeadMessage and returns to normal state:

```
action processHead(nc: NormalCell; net: Net; hm: HeadMessage) is
when hm ∈ net.mess ∧ hm.dst = nc do
  net.mess′ = net.mess - {hm} ∧ nc.next_c′ = hm.new ∧ nc.state′ = normal();
end;
```

Implementation of the delete transaction, which is more complicated, is not described here.

[1]Shadow role means that only shadow attributes of the role are accessed in the action, therefore, it can be left out in implementation.

4.3 Implementation and Testing

Implementation (dli) of DISTRIBUTED LIST was written in Erlang[1]. As explained in Section 3, we are not able to really create the model m_{IUT} but IUT is instrumented to demonstrate its behaviour according the the most concrete specification `messages`. In other words, we observe such behaviours that could have been produced by m_{IUT} and abstracted to behaviours of `messages`. More formally:

$\forall b \in \textit{observed behaviours} : b \in \mathbf{af}^{messages}_{m_{dli}}(\mathbf{af'}^{m_{dli}}_{dli}(beh'(dli)))$

The testing was carried out offline – behaviours were first produced and their correctness were validated afterwards. The actual validation was done with DISCO ANIMATOR [3], a special tool for animating joint-action specifications. To be able to exploit ANIMATOR observing just states is not enough but we must be notified of the executed actions as well.

Execution of the add transaction of the protocol is safe-guarded by a precondition that the next reference of the node in question must point to itself and the node must be in normal operation state. However, should an implementation inadvertantly leave the second requirement unenforced, simply relying on checking the next reference, an error would have been introduced. In that case, initiating a new add transaction in the midst of the previous add transaction, before the next reference is updated, would become possible. A scenario depicting this situation is given in Figure 4, which contains four states and the three actions responsible for state changes. In the initial state the network (net in the scenario) contains `pred` message and the head cell (hc) and both normal cells (nc_1 and nc_2) refer to themselves and operate in `normal` state. First nc_1 wishes to joint the list and, therefore, action `initiateAdd` is executed; then the head cell handles its part of the transaction `add` (action `processAdd`), but before completing the transaction by executing action `processHead` nc_1 attempts to initiate joining again. However, action `initiateAdd` is not enabled in the third state in the specification and the error is detected.

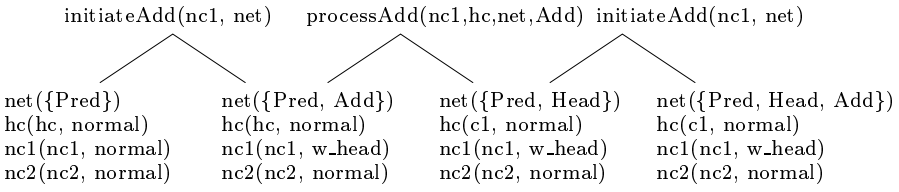


Figure 4: Erroneous scenario.

An actual error in our implementation of the protocol was found during development when the guard of an action was erroneously enabled. This was a result of a reference to an incorrect state in the implementation of the guard. Another error resulted in the consumption of the `pred` message by a cell in the midst of add transaction, never to be put to circulation again. Even though this was a violation of a liveness property, the deadlock was detected by the testing engineer, because no cell was able to commit the delete transaction.

5 Discussion

We have presented a new technique for formal testing of reactive and distributed systems. The technique is based on joint-action specifications, which model behaviours of systems at a high level of abstraction. Testing hypothesis asserts that each IUT has a model m_{IUT} which is indistinguishable from IUT. Now, IUT is correct iff m_{IUT} is obtainable as a legal refinement of the specification for IUT. Correctness is defined so that IUT is correct iff each behaviour induced by m_{IUT} belongs to behaviours induced by its specification. Testing is searching for a counterexample to this relation.

This paper is focused on the ideas behind testing, not how testing is carried out in practise. Thus, this paper reflects virtually no methodological aspects of testing. The next step will be on developing a comprehensive method incorporating these ideas in actual testing. The main contribution of the paper is in giving a formal basis for testing with joint-action specifications. Up to our knowledge joint-action specifications have not been researched from this point of view.

A significant amount of research has been carried out on formal testing methodologies for reactive and distributed systems[9, 15, 11]. A well-known formal approach for conformance testing of concurrent systems is presented by Tretmans in [15]. First a formal framework for testing is developed and then instantiated for *labelled transition systems* (lts). Similarly to our approach formal testing is justified by a *test hypothesis*, which enables the consideration of implementations as formal objects where an implementation relation is used to link a model of an implementation to its specification.

Compared to our method Tretmans emphasizes the input-output relation (where internal state changes are abstracted away) in testing, whereas, our stress is laid on more abstract communication where no distinction is made between input and output. Therefore, our definition for conformance is somewhat different. In [2] Aaltonen et al. have presented a mapping of joint-action specification instances to timed automata, which is an lts-like formalism. This mapping can be utilized for clarifying the differences of the two methods. This is left as future work.

Automatic generation of test suites containing stimulus to IUT with the expected response is a considerably researched topic [6, 15, 14]. For the time being we do not have any algorithms or methods for generating tests. However, we believe that by utilizing observation objectives it is possible to create suitable suites – at least with some guidance by the testing engineer. Presently we are developing a method for producing tests based on observation objectives.

The method for checking that IUT behaves correctly is often called *test oracle*. We have developed a toolset for simulating joint-action specifications [3], which includes COMPILER and ANIMATOR. The compiler produces an engine which maintains the state of a joint-action specification instance and is utilized by the animator. This engine can be considered as the test oracle since it verifies the validity of reached states and execution scenarios. In [7] Dillon and Ramakrishna develop a generic tableau algorithm for generating oracles from temporal logic

specifications.

The abstraction hierarchy of joint-action specifications was not exploited in this paper. In specifying it is considered as the most essential contribution by our method. Therefore, utilizing the hierarchy also in testing is one branch for future work.

To date, research on formalizing testing has been concentrated on finite state formalisms. We are making an important contribution by taking testing to areas where theorem proving has been the dominant mean of ascertaining correctness. However, different approaches have their respective strengths and selection of an approach should be based on the nature of the problem.

References

[1] Open source erlang project WWW page. At `http://www.erlang.org/` on the World Wide Web.

[2] Timo Aaltonen, Mika Katara, and Risto Pitkänen. Verifying real-time joint action specifications using timed automata. In Yulin Feng, David Notkin, and Marie-Claude Gaudel, editors, *16th World Computer Congress 2000, Proceedings of Conference on Software: Theory and Practice*, pages 516–525, Beijing, China, August 2000. IFIP, Publishing House of Electronics Industry and International Federation for Information Processing.

[3] Timo Aaltonen, Mika Katara, and Risto Pitkänen. DisCo toolset – the new generation. *Journal of Universal Computer Science*, 7(1):3–18, 2001. http://www.jucs.org.

[4] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.

[5] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3:73–87, 1989.

[6] Igor Burdonov, Alexander Kossatchev, Alexander Petrenko, and Dmitri Galter. Kvest: Automated generation of test suites from formal specifications. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods: World Congress on Formal Methods in the Development of Computing Systems*, number 1708 in Lecture Notes in Computer Science, pages 608–621. Springer–Verlag, 1999.

[7] Laura K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Foundations of Software Engineering*, pages 106–117, 1996.

[8] Nissim Francez and Ira R Forman. *Interacting Processes – A Multiparty Approach to Coordinated Distributed Programming.* Addison-Wesley, 1996.

[9] Juhana Helovuo and Sari Leppänen. Exploration testing. In *Proceedings of ICACSD 2001, 2nd IEEE International Conference on Application of Concurrency to System Design*, pages 201–210. IEEE Computer Society, June 2001.

[10] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[11] H. Lötzbeyer and A. Pretschner. Testing concurrent reactive systems with constraint logic programming. In *Proceedings 2nd workshop on Rule-Based Constraint Reasoning and Programming*, Singapore, September 2000.

[12] Seungjoon Park and David Dill. Protocol verification by aggregation of distributed transactions. In *Proceedings of the International Conference on Computer-Aided Verification*, Lecture Notes in Computer Science, pages 300–310. Springer-Verlag, July 1996.

[13] Amir Pnueli. The temporal logic of programs. In *In Proceedings of the 18th IEEE Symposium Foundations of Computer Science (FOCS 1977)*, pages 46–57, 1977.

[14] J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.

[15] Jan Tretmans. Testing concurrent systems: A formal approach. In *Proceedings of CONCUR'99 Concurrency Theory*, number 1664 in Lecture Notes in Computer Science, pages 46–65. Springer–Verlag, 1999.

[16] René G. Vries and Jan Tretmans. Towards formal test purposes. In Ed Brinksma and Jan Tretmans, editors, *Proceedings of workshop on Formal Approaches to Testiong of Software 2001*, pages 61–76, August 2001.

Queued Testing of Transition Systems with Inputs and Outputs

Alex Petrenko[a] and Nina Yevtushenko[b]

a - CRIM, Centre de recherche informatique de Montreal,
550 Sherbrooke West, Suite 100, Montreal, H3A 1B9 Canada,
E-mail: Petrenko@crim.ca

b - Tomsk State University, 36 Lenin Str., Tomsk, 634050, Russia,
E-mail: Yevtushenko@elefot.tsu.ru

Abstract The paper studies testing based on input/output transition systems, also known as input/output automata. It is assumed that a tester can never prevent a system under test from producing outputs, while the system does not block inputs from the tester either. Thus, input from the tester and output from the system may occur simultaneously and should be queued in finite buffers between the tester and system. A framework for a so-called queued-quiescence testing is developed, based on the idea that the tester should consist of two test processes, one process is applying inputs via a queue to a system under test and another one is reading outputs from a queue until it detects no more outputs from the system, i.e., the tester detects quiescence in the system. The testing framework is then generalized with a so-called queued-suspension testing by considering a tester that has several pairs of input and output processes. It is demonstrated that such a tester can check finer implementation relations than a queued-quiescence tester. Procedures for test derivation are proposed for a given fault model comprising possible implementations.

1. Introduction

The problem of deriving tests from state-oriented models that distinguish between input and output actions is usually addressed with one of the two basic assumptions about the relationships between inputs and outputs. Assuming that a pair of input and output constitutes an atomic system's action, in other words, that a system cannot accept next input before producing output as a reaction to a previous input, one relies on the input/output Finite State Machine (FSM) model. There is a large body of work on test generation from FSM with various fault models and test architectures, for references see, e.g., [Petr01], [BoPe94]. A system, where next input can arrive even before output is produced in response to a previous input, is usually modeled by the input/output automaton model [LyTu89], also known as the input/output transition system (IOTS) model (the difference between them is marginal, at least from the testing perspective). Compared to the FSM model, this model has received a far less attention in the testing community, see, e.g., [BrTr01], [Phal93], [Sega93]. In this paper, we consider the IOTS model and take a close look on some basic assumptions underlying the existing IOTS testing frameworks.

One of the most important works on test generation from labeled transition systems with inputs and outputs is [Tret96]. In this paper, it is assumed that a tester (implementing a given test case) and an IOTS interact as two labeled transition systems and not as IOTS.

Accordingly, the LTS composition operator used to formalize this interaction does not distinguish between inputs and outputs, so the tester need not be input-enabled to satisfy compatibility conditions for composing IOTS [LyTu89]. The tester seems to be able to preempt output from the system any time it decides to send input to the system. This allows the tester to avoid choosing between inputs and outputs, keeping the test process deterministic. On the other hand, such a tester appears to be able to override the principle that "output actions can never be blocked by the environment" [Tret96, p.106].

Another assumption about the tester is taken by Tan and Petrenko [TaPe98]. In this work, it is recognized that the tester cannot block the system's outputs, it is only assumed that the tester can detect the situation when it offers input to the system, but the latter, instead of consuming it, issues an output (a so-called "exception"). An exception halts a current test run avoiding thus any further non-deterministic behavior and results in the verdict **inconclusive**. Notice that the tester of [Tret96] has only two verdicts, **pass** and **fail**.

Either approach relies on an assumption that is not always justified in a real testing environment. As an example, consider the situation when the tester cannot directly interact with the IUT, because of a context, such as queue or interface, between them. As pointed out in [dVBF02], to apply the test derivation algorithm of [Tret96], one has to take into account the presence of a queue context. It also states "the assumption that we can synthesize every stimulus and analyze every observation is strong", so that some problems in observing quiescence occur.

The case when IOTS is tested via infinite queues is investigated by Verhaard et al [VTKB92]. The proposed approach relies on an explicit combined specification of a given IOTS and queue context, so it is not clear how this approach could be implemented in practice. This context is also considered in [JJTV99], where a stamping mechanism is proposed to order the outputs with respect to inputs, while quiescence is ignored. A stamping process has to be synchronously composed with the IUT as the tester in [Tret96].

We also notice that we are aware of the only work [TaPe98] that uses fault models in test derivation from IOTS. In [Tret96] and [VTKB92], a test case is derived from a trace provided by the user.

The above discussion indicates a need for another approach that does not rely on such strong assumptions about the testing environment and incorporates a fault model to derive tests that can be characterized in terms of fault detection. In this paper, we report on our preliminary findings in attempts to elaborate such an approach. In particular, we introduce a framework for testing IOTS, assuming that a tester can never prevent a system under test from producing outputs, while the system does not block inputs from the tester either and, thus, input and output actions may occur simultaneously and should be queued in finite buffers between the tester and system.

The paper is organized as follows. In Section 2, we introduce some basic definitions and define a composition operator for IOTS based on a refined notion of compatibility of IOTS first defined in [LyTu89]. Section 3 presents our framework for a so-called queued-quiescence testing, based on the idea that the tester should consist of two test processes, one process is applying inputs to a system under test via a finite input queue and another one is reading outputs that the system puts into a finite output queue until it detects no more outputs from the system, i.e., the tester detects quiescence in the system. We elaborate such a tester and formulate several implementation relations that can be tested with a queued-quiescence tester. In Section 4, we discuss how queued-quiescence tests can be derived for a given specification and fault model that comprises a finite set of implementations. In Section 5, we generalize our testing framework with a so-called queued-suspension testing by

allowing a tester to have several pairs of input and output processes and demonstrate that a queued-suspension tester can check finer implementation relations than a queued-quiescence tester. We conclude by comparing our contributions with previous work and discussing further work.

2. Preliminaries

A *labeled transition system*, or simply a labeled transition system (LTS), is a 4-tuple $L = <S, \Sigma, \lambda, s_0>$, where S is a finite non-empty set of states with the initial state s_0; Σ is a finite set of actions; $\lambda \in S \times \Sigma \times S$ is a transition relation. In this paper, we consider only LTS such that $(s, a, s'), (s, a, s'') \in \lambda$ implies $s' = s''$. These are deterministic LTS.

Let $L_1 = <S, \Sigma_1, \lambda_1, s_0>$ and $L_2 = <T, \Sigma_2, \lambda_2, t_0>$, the *parallel composition* $L_1 \| L_2$ is defined as the LTS $<R, \Sigma_1 \cup \Sigma_2, \lambda, s_0t_0>$, where the set of states $R \subseteq S \times T$ and the transition relation λ are smallest sets obtained by application of the following inference rules:

- if $a \in \Sigma_1 \cap \Sigma_2$, $(s, a, s') \in \lambda_1$, and $(t, a, t') \in \lambda_2$ then $(st, a, s't') \in \lambda$;
- if $a \in \Sigma_1 \backslash \Sigma_2$, $(s, a, s') \in \lambda_1$, then $(st, a, s't) \in \lambda$;
- if $a \in \Sigma_2 \backslash \Sigma_1$, $(t, a, t') \in \lambda_1$, then $(st, a, st') \in \lambda$.

We use the LTS model to define a transition system with inputs and outputs. The difference between these two types of actions is that no system can deny an input action from its environment, while this is completely up to the system when to produce an output, so the environment cannot block the output. Formally, an *input/output transition system* (IOTS) L is a LTS in which the set of actions Σ is partitioned into two sets, the set of input actions I and the set of output actions O. Given state s of L, we further denote $init(s)$ the set of actions defined at s, i.e. $init(s) = \{a \in \Sigma \mid \exists s' \in S ((s, a, s') \in \lambda)\}$. The IOTS is *input-enabled* if each input action is enabled at any state, i.e., $I \subseteq init(s)$ for each s. State s of the IOTS is called *unstable* if there exists $o \in O$ such that $o \in init(s)$. Otherwise, state is *stable*. A sequence $a_1 \ldots a_k$ over the set Σ is called a *trace* of L in state s if there exist states $s_1, \ldots, s_{k+1}$ such that $(s_i, a_i, s_{i+1}) \in \lambda$ for all $i = 1, \ldots, k$ and $s_1 = s$. We use $traces(s)$ to denote the set of traces of L in state s. Following [Vaan91] and [Tret96], we refer to a trace that takes the IOTS from a given state to a stable state as to a *quiescent* trace.

To define a composition of IOTS, we first state compatibility conditions that define when two IOTS can be composed by relaxing the original conditions of [LyTu89]. Note that $L_1 \| L_2$ for IOTS L_1 and L_2 means the synchronous parallel composition of LTS that are obtained from IOTS by neglecting the difference between inputs and outputs, so these IOTS are treated as LTS.

Definition 1. Let $L_1 = <S, \Sigma_1, \lambda_1, s_0>$, $\Sigma_1 = I_1 \cup O_1$, and $L_2 = <T, \Sigma_2, \lambda_2, t_0>$, $\Sigma_2 = I_2 \cup O_2$, be two IOTS such that the sets $I_1 \cap I_2$ and $O_1 \cap O_2$ are empty. Let st be a state of the composition $L_1 \| L_2$. The L_1 and L_2 are *compatible in state st* if

- $a \in init(s)$ implies $a \in init(t)$ for any $a \in I_2 \cap O_1$ and
- $a \in init(t)$ implies $a \in init(s)$ for any $a \in I_1 \cap O_2$.

The L_1 and L_2 are said to be *compatible* if they are compatible in the state s_0t_0; otherwise they are *incompatible*. L_1 and L_2 are *fully compatible* if they are compatible in all the states of the composition $L_1 \| L_2$.

Clearly, two input-enabled IOTS with $I_1 = O_2$ and $I_2 = O_1$ are fully compatible, but the converse is not true. Based on the notion of compatibility we define what we mean by a

parallel composition of two IOTS. Let $IOTS(I, O)$ denote the set of all possible IOTS over the input set I and output set O.

Definition 2. The *composition operator*][: $IOTS(I_1, O_1) \times IOTS(I_2, O_2) \rightarrow IOTS((I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2)$, where the sets $I_1 \cap I_2$ and $O_1 \cap O_2$ are empty, is defined as follows. Let $L_1 = \langle S, \Sigma_1, \lambda_1, s_0 \rangle$, $\Sigma_1 = I_1 \cup O_1$, and $L_2 = \langle T, \Sigma_2, \lambda_2, t_0 \rangle$, $\Sigma_2 = I_2 \cup O_2$ be compatible IOTS. If L_1 and L_2 are compatible in state st of the composition $L_1 \parallel L_2$, then $st -a\rightarrow s't'$ in $L_1 \parallel L_2$ implies the same transition in L_1][L_2. If L_1 and L_2 are incompatible in state st, then there are no outgoing transitions from the state in L_1][L_2, i.e., st is a deadlock.

The IOTS L_1][L_2 can be obtained from the LTS $L_1 \parallel L_2$ by pruning outgoing transitions from states where the IOTS L_1 and L_2 are not compatible. For fully compatible IOTS, the results of both operators, || and][, coincide.

3. Framework for Queued-Quiescence Testing of IOTS

In a typical testing framework, it is usually assumed that the two systems, an implementation under test (IUT) and tester, form a closed system. This means that if L_1 is a tester, while L_2 is an IOTS modeling the IUT, then $\Sigma_1 = \Sigma_2$, $I_1 = O_2$, and $I_2 = O_1$. To be compatible with any IOTS over the given alphabet $\Sigma_2 = I_2 \cup O_2$ the tester should be input-enabled. However, the input-enableness has two implications on a behavior of the tester. Its behavior becomes infinite since inputs enabled in each state create cycles and non-deterministic since the tester has to choose non-deterministically between input and output. Both features are usually considered undesirable. Testers should be deterministic and have no cycles. The two requirements are contradicting.

It turns out that a tester processing outputs of an IUT separately from inputs could meet both requirements. To achieve this, it is sufficient to decompose the tester into two processes, one for inputs and another for outputs. Intuitively, this could be done as follows. The input test process only sends to the IUT via input buffer a given (finite) number of consecutive test stimuli. In response to the submitted input sequence, the IUT produces outputs that are stored in another (output) buffer. The output test process, that is simply an observer, only accepts outputs of the IUT by reading the output buffer. All the output sequences the specification IOTS can produce in response to the submitted input sequence, should take the output test process into terminal states labeled with the verdict **pass**, while any other output sequence produced by an IUT should take the output test process to a terminal state labeled with the verdict **fail**. Since the notion of a tester is based on the definition of a set of output sequences that the specification IOTS can produce in response to a submitted input sequence, we formalize both notions as follows.

Let L be an IOTS defined over the action set $\Sigma = I \cup O$ and $pref(\alpha)$ denote the set of all the prefixes of a sequence α over the set Σ. The set $pref(\alpha)$ has the empty sequence ε. Also given a set $P \subseteq \Sigma^*$, let $\{\beta \in pref(\gamma) \mid \gamma \in P\} = pref(P)$.

Definition 3. Given an input word $\alpha \in I^*$, the *input test process* with α is a tuple $\alpha = \langle pref(\alpha), \varnothing, I, \lambda_\alpha, \varepsilon \rangle$, where the set of inputs is empty, while the set of outputs is I, $\lambda_\alpha = \{(\beta, a, \beta a) \mid \beta a \in pref(\alpha)\}$, and the initial state is ε.

We slightly abuse α to denote both, the input sequence and the input test process that executes this sequence. It is easy to see that each input test process is fully compatible with any IOTS L that is input-enabled and defined over the set of inputs I. Notice that in this paper, we consider only input-enabled IOTS specifications, while an implementation IOTS (that models an IUT) is always assumed to be input-enabled.

To define an output test process that complements an input test process α, we have first to determine all the output sequences, valid and invalid, the output test process has to expect from IUT. The number of valid output sequences becomes infinite when the specification oscillates, in other words, when it has cycles that involve only outputs. Further, we always assume that the specification IOTS $Spec = <S, I \cup O, \lambda, s_0>$ does not possess this property. Thus, in response to α, the IOTS *Spec* can execute any trace that is a completed trace [Glab90] of the IOTS α][*Spec* leading into a terminal state, i.e., into state g, where $init(g) = \varnothing$. Let $ctraces(\alpha$][*Spec*) be the set of all such traces. It turns out that the set $ctraces(\alpha$][*Spec*) is closely related to the set of quiescent traces of the specification $qtraces(Spec)$, viz. it includes each quiescent trace β whose input projection, denoted $\beta_{\downarrow I}$, is the sequence α.

Proposition 4. $ctraces(\alpha$][*Spec*$) = \{\beta \in qtraces(Spec) \mid \beta_{\downarrow I} = \alpha\}$.

Thus, the set $ctraces(\alpha$][*Spec*$)_{\downarrow O} = \{\beta_{\downarrow O} \mid \beta \in qtraces(Spec)$ & $\beta_{\downarrow I} = \alpha\}$ contains all the output sequences that can be produced by the *Spec* in response to the input sequence α.

Let $qtraces(s)$ be the set of quiescent traces of *Spec* in state s. Given a quiescent trace $\beta \in qtraces(s)$, the sequence $\beta_{\downarrow I}\beta_{\downarrow O}\delta$ is said to be a *queued-quiescent trace* of *Spec* in state s, where $\delta \notin \Sigma$ is a designated symbol that denotes the absence of outputs, i.e., quiescence. We use $Qqtraces(s)$ to denote the set of queued-quiescence traces of s $\{(\beta_{\downarrow I}\beta_{\downarrow O}\delta) \mid \beta \in qtraces(s)\}$ and $Qqtraces_o(s, \alpha)$ to denote the set $\{\beta_{\downarrow O}\delta \mid \beta \in qtraces(s)$ & $\beta_{\downarrow I} = \alpha\}$. Next, we define the output test process itself.

Given the input test process α and the set $Qqtraces_o(s_0, \alpha)$, we define a set of output sequences $out(\alpha)$ the output test process can receive from an IUT. Intuitively, it is sufficient to consider all the shortest invalid output sequences along with all valid ones. Any valid sequence should not be followed by any further output action, as the specification becomes quiescent, while any premature quiescence indicates that the observed sequence is not a valid output sequence. The set $out(\alpha)$ is defined as follows. For each $\beta \in pref(Qqtraces_o(s_0, \alpha))$ the sequence $\beta \in out(\alpha)$ if $\beta \in Qqtraces_o(s_0, \alpha)$, otherwise $\beta a \in out(\alpha)$ for all $a \in O \cup \{\delta\}$ such that $\beta a \notin pref(Qqtraces_o(s_0, \alpha))$.

Definition 5. The *output test process* for the IOTS *Spec* and the input test process α is a tuple $<pref(out(\alpha)), O \cup \{\delta\}, \varnothing, \lambda_\alpha, \varepsilon>$, where $pref(out(\alpha))$ is the state set, $O \cup \{\delta\}$ is the input set, the output set is empty, $\lambda_\alpha = \{(\beta, a, \beta a) \mid \beta a \in pref(out(\alpha))\}$ and ε is the initial state. State $\beta \in pref(out(\alpha))$ is labeled with the verdict **pass** if $\beta \in Qqtraces_o(s_0, \alpha)$ or with the verdict **fail** if $\beta \in out(\alpha) \backslash Qqtraces_o(s_0, \alpha)$.

We reuse $out(\alpha)$ to denote the output test process that complements the input test process α. For a given input sequence $\alpha \in I^*$ the pair $(\alpha, out(\alpha))$ is called a *queued-quiescence tester* or *test case*.

To describe the way the output tester interacts with an IOTS $Imp \in IOTS(I, O)$ after the input test process α has terminated its execution against *Imp*, we denote $(\alpha$][*Imp*$)_{\downarrow O,\delta}$ the

IOTS that is obtained from (α][*Imp*) by first projecting it onto the output alphabet *O* and subsequent augmenting all the stable states of the resulting projection by self-looping transitions labeled with δ. In doing so, we treat the symbol δ as an input of the output test process, assuming that the tester synchronizing on δ just detects the fact that its buffer has no more symbols to read. Strictly speaking, treated as an output, a repeated δ violates the compatibility of a system in a stable state and a tester that reaches its terminal state. With this in mind, the correctness of the construction of the output test process (the soundness of the tester) can be stated as follows.

Proposition 6. For any *Imp* ∈ *IOTS*(*I*, *O*) if the IOTS (α][*Imp*)$\downarrow_{O,\delta}$][*out*(α) reaches a state where (α][*Imp*)$\downarrow_{O,\delta}$ and *out*(α) are incompatible, then the output tester *out*(α) is in a terminal state. For the *Spec* a state, where (α][*Spec*)$\downarrow_{O,\delta}$ and *out*(α) are incompatible is reached only after quiescence, while the output tester is in a terminal state labeled with the verdict **pass**.

Thus, the tester composed of two independent processes meets both requirements, namely, it is compatible in all the states, save for the terminal ones, with any IOTS, it has no cycles and never need choosing between input and output, thus the tester possesses the required properties.

The composition (α][*Imp*)$\downarrow_{O,\delta}$][*out*(α) of a queued-quiescence tester for a given specification with an implementation IOTS *Imp* over the same action set as the specification has one or several terminal states. In a particular test run, one of these states with the verdict **pass** or **fail** is reached. Considering the distribution of verdicts in the terminal states of the composition, the three following cases are possible:

Case 1. All the states have **fail**.

Case 2. States have **pass** as well as **fail**.

Case 3. All the states have **pass**.

These cases lead us to various relations between an implementation and the specification that can be established by the queued-quiescence testing.

In the first case, the implementation is distinguished from the specification in a single test run.

Definition 7. Given IOTS *Spec* and *Imp*, *Imp* is *queued-quiescence separable* from *Spec*, if there exists a test case (α, *out*(α)) for *Spec* such that the terminal states of the IOTS (α][*Imp*)$\downarrow_{O,\delta}$][*out*(α) are labeled with the verdict **fail**.

In the second case, the implementation can also be distinguished from the specification provided that a proper run is taken by the implementation during the test execution.

Definition 8. Given IOTS *Spec* and *Imp*, *Imp* is *queued-quiescence distinguishable* from *Spec*, if there exists a test case (α, *out*(α)) for *Spec* such that the terminal states of (α][*Imp*)$\downarrow_{O,\delta}$][*out*(α) are labeled with the verdicts **pass** and **fail**.

Consider now case 3, when for a given test case (α, *out*(α)) all the states have **pass**. In this case, the implementation does nothing illegal when the test case is executed, as it produces only valid output sequences. Two situations can yet be distinguished here. Either there exists a **pass** state of the output test process that is not included in any terminal state of

(α][*Imp*)$_{\downarrow O,\delta}$][*out*(α) or there is no such a state. The difference is that with the given test case in the former situation, the implementation could still be distinguished from its specification, while in the latter, it could not. This motivates the following definition.

Definition 9. Given IOTS *Spec* and *Imp*,

- *Imp* is said to be *queued-quiescence trace-included* in the *Spec* if for all $\alpha \in I^*$ no terminal state of the IOTS (α][*Imp*)$_{\downarrow O,\delta}$][*out*(α) is labeled with the verdict **fail**.
- *Imp* and *Spec* are *queued-quiescence trace-equivalent* if for all $\alpha \in I^*$ all the terminal states of the IOTS (α][*Imp*)$_{\downarrow O,\delta}$][*out*(α) include all the **pass** states of *out*(α) and only them.
- *Imp* that is queued-quiescence trace-included in the *Spec* but not queued-quiescence trace-equivalent to the *Spec* is said to be *queued-quiescence weakly-distinguishable* from *Spec*.

We characterize the above relations in terms of queued-quiescent traces.

Proposition 10. Given IOTS *Spec* with the initial state s_0 and *Imp* with the initial state t_0,

- *Imp* is queued-quiescence separable from *Spec* iff there exists an input sequence α such that $Qqtraces_o(t_0, \alpha) \cap Qqtraces_o(s_0, \alpha) = \varnothing$.
- *Imp* that is not queued-quiescence separable from *Spec* is queued-quiescence distinguishable from it iff there exists an input sequence α such that $Qqtraces_o(t_0, \alpha) \notin Qqtraces_o(s_0, \alpha)$.
- *Imp* that is not queued-quiescence distinguishable from *Spec* is queued-quiescence weakly-distinguishable from it iff there exists an input sequence α such that $Qqtraces_o(t_0, \alpha) \subset Qqtraces_o(s_0, \alpha)$.
- *Imp* is queued-quiescence trace-included into *Spec*, iff $Qqtraces(t_0) \subseteq Qqtraces(s_0)$.
- *Imp* and *Spec* are queued-quiescence trace-equivalent iff $Qqtraces(t_0) = Qqtraces(s_0)$.

Figure 1 provides an example of IOTS that are not quiescent trace equivalent, but are queued-quiescence trace-equivalent. Indeed, the quiescent trace $aa1\delta$ of the IOTS L_2 is not a trace of the IOTS L_1. In both, the input sequence *a* yields the queued-quiescent trace $a1\delta$, *aa* yields the queued-quiescent traces $aa1\delta$ and $aa2\delta$, any longer input sequence results in the same output sequences as *aa*.

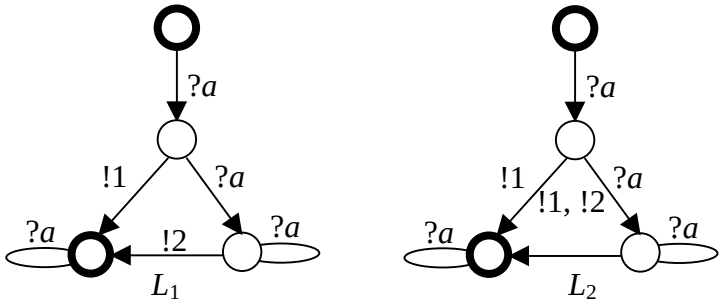


Figure 1: Two IOTS that have the different sets of quiescent traces, but are queued-quiescence trace-equivalent; inputs are decorated with "?", outputs with "!"; stable states are depicted in bold.

The IOTS L_1 and L_2 are considered indistinguishable in our framework, while according to the **ioco** relation [Tret96], they are distinguishable. The IOTS L_2 has the quiescent trace $aa1$ that is not a trace of L_1, therefore, to distinguish the two system, the tester has to apply two consecutive inputs a. The output 1 appearing only after the second input a indicates that the system being tested is, in fact, L_2 and not L_1. However, to make such a conclusion, the tester should be able to prevent the appearance of the output 1 after the first input a. Under our assumption, it is not possible. The tester interacts with the system via queues and has no way of knowing when the output is produced. The presence of a testing context that is a pair of finite queues in our case, makes implementation relations that could be tested via the context coarser, as is usually the case [PYBD96].

4. Deriving Queued-Quiescence Test Cases

Proposition 10 indicates the way test derivation could be performed for the IOTS *Spec* and an explicit fault model when we are given a finite set of implementations. Namely, for each *Imp* in the fault model, we may first attempt to determine an input sequence α such that $Qqtraces_o(t_0, \alpha) \cap Qqtraces_o(s_0, \alpha) = \varnothing$. If fail we could next try to find α such that $Qqtraces_o(t_0, \alpha) \not\subseteq Qqtraces_o(s_0, \alpha)$. If $Qqtraces_o(t_0, \alpha) \subseteq Qqtraces_o(s_0, \alpha)$ for each α the question is about an input sequence α such that $Qqtraces_o(t_0, \alpha) \neq Qqtraces_o(s_0, \alpha)$, thus $Qqtraces_o(t_0, \alpha) \subset Qqtraces_o(s_0, \alpha)$. Based on the found input sequence, a queued-quiescence test case for the *Imp* in hand can be constructed, as explained in the previous section. If no input sequence with this property can be determined we conclude that the IOTS *Spec* and *Imp* are queued-quiescence trace-equivalent, they cannot be distinguished by the queued-quiescence testing.

Search for an appropriate input sequence could be performed in a straightforward way by considering input sequences of increasing length. To do so, we just parameterize Definitions 7, 8 and 9 and accordingly Proposition 10 with the length of input sequences. Given a length of input sequences k, let $Qqtraces^k(s_0) = \{(\beta_{\downarrow I}\beta_{\downarrow O}\delta) \mid \beta \in qtraces(s_0) \ \&\ |\beta_{\downarrow I}| \leq k\}$. The set $Qqtraces^k(s_0)$ is finite for the IOTS L with a finite set of quiescence traces. Then, e.g., *Imp* and *Spec* are queued-quiescence k-trace-equivalent iff $Qqtraces^k(t_0) = Qqtraces^k(s_0)$. If length of α such that $Qqtraces_o(t_0, \alpha) \not\subseteq Qqtraces_o(s_0, \alpha)$ is k then *Imp* is said to be queued-quiescence k-distinguishable from *Spec*. With these parameterized definitions, we examine all the input sequences starting from an empty input action. The procedure terminates when the two IOTS are distinguished or when the value of k reaches a predefined maximum defined by the input buffer of the IUT available for queued testing.

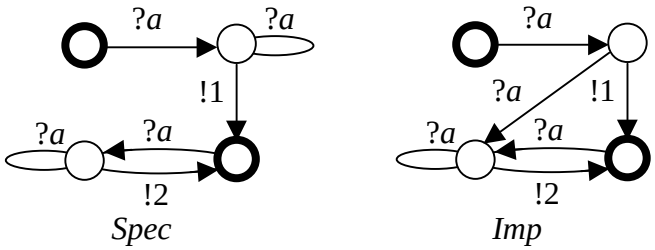
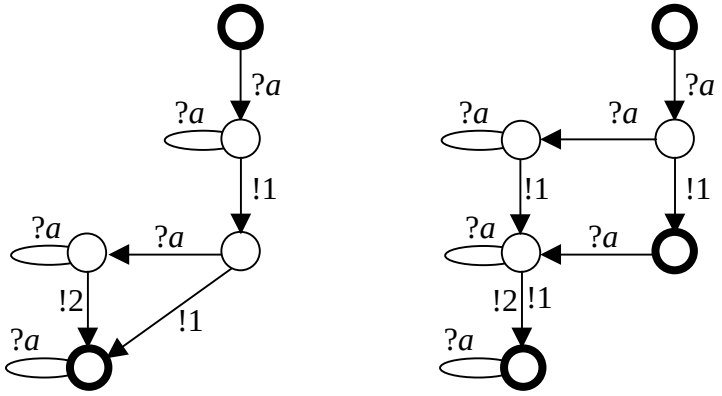


Figure 2: The IOTS that are queued-quiescence 2-distinguishable, but not queued-quiescence 1-distinguishable.

Consider the example in Figure 2. By direct inspection, one can assure *Imp* is not queued-quiescence 1-distinguishable from *Spec*, for both produce the output 1 in response to the

input *a*. However, it is queued-quiescence 2-distinguishable from *Spec*. Indeed, in response to the sequence ?*a*?*a* the *Spec* can produce the output 1 or 12. While the *Imp* - 2 or 12.
It is interesting to notice that the notion of *k*-distinguishability applied to the IOTS and FSM models exhibits different properties. In particular, two *k*-distinguishable FSM are also *k*+1-distinguishable. This does not always hold for IOTS. The system *Imp* in Figure 3 is queued-quiescence 1-distinguished from *Spec*; however, it is not queued-quiescence *k*-distinguished from *Spec* for any $k > 1$.



Spec *Imp*

Figure 3: The IOTS that are queued-quiescence 1-distinguishable, but not queued-quiescence *k*-distinguishable for $k > 1$.

This indicates that a special care has to be taken when one attempts to adapt FSM-based methods to the queued testing of IOTS.

There is at least one special case when distinguishability can be decided without determining the sets of queued-quiescence *k*-traces for various *k*.

Let $Imp_{\downarrow O}$ denote the set of output projections of all traces of *Imp*.

Proposition 11. Given two IOTS *Spec* and *Imp*, if the set $Imp_{\downarrow O}$ is not a subset of $Spec_{\downarrow O}$ then *Imp* is queued-quiescence distinguishable from *Spec*, moreover, any quiescence trace $\beta \in qtraces(Spec)$ such that $\beta_{\downarrow O} \in Imp_{\downarrow O} \backslash Spec_{\downarrow O}$ yields a queued-quiescence test case $(\beta_{\downarrow I}, out(\beta_{\downarrow I}))$ that when executed against the *Imp* produces the verdict **fail**.

The statement suggests a procedure for deriving test cases.

Procedure for deriving a test case that distinguishes the IOTS *Imp* from *Spec*.
Input: IOTS *Spec* and *Imp* such that the set $Imp_{\downarrow O}$ is not a subset of $Spec_{\downarrow O}$.

Output: A queued-quiescence test case $(\alpha, out(\alpha))$ such that $Qqtraces_o(t_0, \alpha) \not\subseteq Qqtraces_o(s_0, \alpha)$.

Step 1. By use of a subset construction, project *Imp* and *Spec* onto the output set *O*.

Step 2. Using the direct product of the obtained projections, determine a trace ρ that is a trace of the output projection of *Imp* while not being a trace of that of *Spec*.

Step 3. Compose LTS $<pref(\rho), O, \lambda_\rho, \varepsilon>$ with the *Imp* and obtain the LTS, where each trace is a trace of the *Imp* with the output projection ρ. Determine the input projection α of any trace of the obtained LTS and the queued-quiescence test case $(\alpha, out(\alpha))$.

Proposition 12. Given two IOTS *Spec* and *Imp*, let (α, *out*(α)) be the queued-quiescence test case derived by the above procedure. Then the queued-quiescence test case executed against the *Imp* produces the verdict **fail**. Moreover, if no **pass** verdict can be produced then *Imp* is queued-quiescence separable from *Spec*.

5. Queued-Suspension Testing of IOTS

In the previous sections, we explored the possibilities for distinguishing IOTS based on their queued-quiescent traces. The latter are pairs of input and output projections of quiescent traces. If systems with different quiescent traces have the same set of queued-quiescent traces, no queued-quiescence test case can differentiate them. However, sometimes such IOTS can still be distinguished by a queued testing, as we demonstrate below.

Consider the example in Figure 4. Here the two IOTS have different sets of quiescent traces, however, the have the same set of queued-quiescent traces {$a1\delta$, $aa1\delta$, $aa12\delta$, $aaa1\delta$, $aaa12\delta$, … }. In the testing framework presented in Section 3, they are not distinguishable. Indeed, we cannot tell them apart when a single input is applied to their initial states. Moreover, in response to the input sequence *aa* and to any longer sequence, they produce the same output sequence 12. The difference is that IOTS *Imp*, while producing the output sequence 12, becomes quiescent just before the output 2 and the IOTS *Spec* does not. The problem is that this quiescence is not visible through the output queue by the output test process that expects either 1 or 12 in response to *aa*. The queued-quiescence tester can detect the quiescence after reading the output sequence 12 as an empty queue, but it cannot detect an "intermediate" quiescence of the system. It has no way of knowing whether the system becomes quiescent before a subsequent input is applied. Both inputs are in the input buffer and it is completely up to the system when to read the second input.

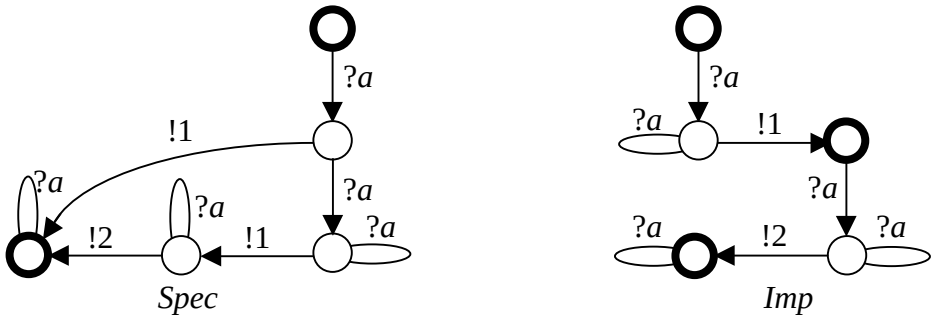


Figure 4: The queued-quiescence equivalent IOTS.

Intuitively, further decomposing the tester for the *Spec* into two input and two output test processes could solve the problem. In this case, testing is performed as follows. The first input test process issues the input *a*. The first output test process expects the output 1 followed by a quiescence δ, when the quiescence is detected, the control is transferred to the second input test process that does the final *a*. Then the second output test process expects quiescence. If, instead, it detects the output 2 it produces the verdict **fail** which indicates that the IUT is *Imp* and not *Spec*. Opposed to a queued-quiescence tester, such a tester can detect an intermediate quiescence of the system. The example motivates the following definitions.

Let $\alpha_1 \ldots \alpha_p$ be a finite sequence of input words such that $\alpha_i \in I^*$, $i = 1, \ldots, p$, and $\alpha_i \neq \varepsilon$ for $i \neq 1$. Each word α_i defines the input test process α_i (see Definition 3). To define output test processes that complement input test processes $\alpha_1 \ldots \alpha_p$, we first notice that the first

output test process is designed based on the fact that the *Spec* starts in the only state that is its initial state. This is no longer true for any subsequent output test process; the *Spec* can start in one of several stable states, depending on an output sequence it has produced in response to the stimuli from the previous input test processes. Let $\alpha\beta\delta \in Qqtraces(s_0)$, then we use *Spec*-**after**-(α, β) to denote the set of stable states that are reached by *Spec* when it executes all possible quiescent traces with the input projection α and output projection β. Consider the input test process α_2, before it starts, the *Spec* has produced one of $|Qqtraces_o(s_0, \alpha_1)|$ output sequences, each of which defines the set of stable states *Spec*-**after**-(α_1, β), where $\beta \in Qqtraces_o(s_0, \alpha_1)$, these are the starting states for the second input test process α_2. Thus, we have to define $|Qqtraces_o(s_0, \alpha_1)|$ output test processes that complement the input test process α_2. Each valid output sequence produced by the IUT so far is used to decide which pair of input and output test processes should execute next, while each invalid sequence terminates the test execution. Thus, testing becomes adaptive. For an output sequence $\beta \in Qqtraces_o(s_0, \alpha_1)$, the *Spec* can produce in response to α_2 any output sequence in the set $Qqtraces_o(Spec\text{-}\mathbf{after}\text{-}(\alpha_1, \beta), \alpha_2)$. Generalizing this to α_{i+1}, we have the following. The set of output sequences the *Spec* can produce in response to α_{i+1} is $Qqtraces_o(Spec\text{-}\mathbf{after}\text{-}(\alpha_1\ldots\alpha_i, \beta_1\ldots\beta_i), \alpha_{i+1})$, where for the output projection $\beta_1\ldots\beta_i$ it holds that $\beta_1 \in Qqtraces_o(s_0, \alpha_1)$ and $\beta_j \in Qqtraces_o(Spec\text{-}\mathbf{after}\text{-}(\alpha_1\ldots\alpha_{j-1}, \beta_1\ldots\beta_{j-1}), \alpha_j)$ for each $j = 2, \ldots, i$. A sequence of output words $\beta_1\ldots\beta_i$ with such a property is said to be *consistent* (with the corresponding sequence of input words $\alpha_1\ldots\alpha_i$).

Each output projection $\beta_1\ldots\beta_i$ defines, therefore, a distinct test output process for the (i+1)-*th* input test process. The set of output sequences $out(\alpha_{i+1}, \beta_1\ldots\beta_i)$ the output test process for the given sequence $\beta_1\ldots\beta_i$ can receive from an IUT is defined as follows. For each $\gamma \in pref(Qqtraces_o(Spec\text{-}\mathbf{after}\text{-}(\alpha_1\ldots\alpha_i, \beta_1\ldots\beta_i), \alpha_{i+1})$ the sequence $\gamma \in out(\alpha_{i+1}, \beta_1\ldots\beta_i)$ if $\gamma \in Qqtraces_o(Spec\text{-}\mathbf{after}\text{-}(\alpha_1\ldots\alpha_i, \beta_1\ldots\beta_i), \alpha_{i+1})$, otherwise $\gamma a \in out(\alpha_{i+1}, \beta_1\ldots\beta_i)$ for all $a \in O \cup \{\delta\}$ such that $\gamma a \notin pref(Qqtraces_o(Spec\text{-}\mathbf{after}\text{-}(\alpha_1\ldots\alpha_i, \beta_1\ldots\beta_i), \alpha_{i+1})$.

Let $\beta_1\ldots\beta_i{\cdot}pref(out(\alpha_{i+1}, \beta_1\ldots\beta_i))$ denote the set $\{\beta_1\ldots\beta_i\alpha \mid \alpha \in pref(out(\alpha_{i+1}, \beta_1\ldots\beta_i))\}$. Now we are ready to generalize the definition of output test processes (Definition 5), taking into account a valid output sequence produced by an IUT with the preceding test processes.

Definition 13. Let $\alpha_1 \ldots \alpha_{i+1}$ be a sequence of input words and $\beta_1\ldots\beta_i$ be a consistent sequence of output words. An *output test process* for the IOTS *Spec*, sequence $\beta_1\ldots\beta_i$ and the input test process α_{i+1} is a tuple $<\beta_1\ldots\beta_i{\cdot}pref(out(\alpha_{i+1}, \beta_1\ldots\beta_i)), O \cup \{\delta\}, \varnothing, \lambda_{\alpha i+1,\, \beta 1\ldots\beta i}, \beta_1\ldots\beta_i>$, where $\beta_1\ldots\beta_i{\cdot}pref(out(\alpha_{i+1}, \beta_1\ldots\beta_i))$ is the state set, $O \cup \{\delta\}$ is the input set, the output set is empty, $\lambda_{\alpha i+1,\, \beta 1\ldots\beta i} = \{(\beta_1\ldots\beta_i\gamma, a, \beta_1\ldots\beta_i\gamma a) \mid \gamma a \in pref(out(\alpha_{i+1}, \beta_1\ldots\beta_i))\}$ and $\beta_1\ldots\beta_i$ is the initial state. Each state $\beta_1\ldots\beta_i\gamma$ is labeled with the verdict **pass** if $\gamma \in Qqtraces_o(Spec\text{-}\mathbf{after}\text{-}(\alpha_1\ldots\alpha_i, \beta_1\ldots\beta_i), \alpha_{i+1})$ or with the verdict **fail** if $\gamma \in out(\alpha_{i+1}, \beta_1\ldots\beta_i) \backslash Qqtraces_o(Spec\text{-}\mathbf{after}\text{-}(\alpha_1\ldots\alpha_i, \beta_1\ldots\beta_i), \alpha_{i+1})$.

We use $out(\alpha_{i+1}, \beta_1\ldots\beta_i)$ to denote an output test process that complements the input test process α_{i+1} and the output sequence $\beta_1\ldots\beta_i$. For a given sequence of input words $\alpha_1\ldots\alpha_p$, the set of the tuples of pairs $(\alpha_1, out(\alpha_1)), \ldots, (\alpha_p, out(\alpha_p, \beta_1\ldots\beta_{p-1}))$ for all consistent output sequences $\beta_1\ldots\beta_{p-1}$ is called a *queued-suspension tester* or a *queued-suspension test case* and is denoted $(\alpha_1\ldots\alpha_p, Out(\alpha_1\ldots\alpha_p))$.

It is clear that for a single input test process, a queued-suspension tester reduces to a queued-quiescence tester. The queued-suspension testing is more discriminative than

queued-quiescence testing, as Figure 4 illustrates. In fact, consider a queued-quiescence tester derived from a single sequence $\alpha_1 \ldots \alpha_p$ and a queued-suspension tester derived from the sequence of p words α_1, …, α_p, the former uses just the output projection of quiescent traces that have the input projection $\alpha_1 \ldots \alpha_p$ while the latter additionally partitions the quiescent traces into p quiescent sub-traces. Then the two systems that cannot be distinguished by the queued-suspension testing have to produce the same output projection, moreover, the output projections have to coincide up to the partition defined by the partition of the input sequence. This leads us to the notion of queued-suspension traces.

Given a finite sequence of finite input words $\alpha_1 \ldots \alpha_p$, a sequence of queued-quiescence traces $(\alpha_1\beta_1\delta)\ldots(\alpha_p\beta_p\delta)$ is called a *queued-suspension trace* of *Spec* if $\alpha_1\beta_1\delta \in Qqtraces(s_0)$ and for each $i = 2, \ldots, p$ it holds that $\beta_i\delta \in Qqtraces_o(Spec\text{-}\textbf{\textit{after}}\text{-}(\alpha_1 \ldots \alpha_{i-1}, \beta_1 \ldots \beta_{i-1}), \alpha_i)$. We use $Qstraces(s)$ to denote the set of queued-suspension traces of *Spec* in state s.

We define the relations that can be characterized with queued-suspension testing by adapting Definitions 7, 8, and 9.

Definition 14. Given IOTS *Spec* and *Imp*,

- *Imp* is *queued-suspension separable* from *Spec*, if there exist a test case $(\alpha_1 \ldots \alpha_p, Out(\alpha_1 \ldots \alpha_p))$ for *Spec* such that for any consistent output sequence $\beta_1 \ldots \beta_{p-1}$ the terminal states of the IOTS $(\alpha_p \,][\, Imp\ \textbf{after}(\alpha_1 \ldots \alpha_{p-1}, \beta_1 \ldots \beta_{p-1}))\downarrow_{O,\delta} \,][\, out(\alpha_p, \beta_1 \ldots \beta_{p-1})$ are labeled with the verdict **fail**.
- *Imp* is *queued-suspension distinguishable* from *Spec*, if there exist test case $(\alpha_1 \ldots \alpha_p, Out(\alpha_1 \ldots \alpha_p))$ for *Spec* and consistent output sequence $\beta_1 \ldots \beta_{p-1}$ such that the terminal states of the IOTS $(\alpha_p \,][\, Imp\ \textbf{after}(\alpha_1 \ldots \alpha_{p-1}, \beta_1 \ldots \beta_{p-1}))\downarrow_{O,\delta} \,][\, out(\alpha_p, \beta_1 \ldots \beta_{p-1})$ are labeled with the verdicts **pass** and **fail**.
- *Imp* is said to be *queued-suspension trace-included* in the *Spec* if for all $\alpha \in I^*$ and all possible partitions of α into words α_1, …, α_p, no terminal state of IOTS $(\alpha_p \,][\, (\alpha_p \,][\, Imp\ \textbf{after}(\alpha_1 \ldots \alpha_{p-1}, \beta_1 \ldots \beta_{p-1}))\downarrow_{O,\delta} \,][\, out(\alpha_p, \beta_1 \ldots \beta_{p-1})$ is labeled with the verdict **fail**.
- *Imp* and *Spec* are *queued-suspension trace-equivalent* if for all $\alpha \in I^*$, all possible partitions of α into words α_1, …, α_p, and all consistent output sequence $\beta_1 \ldots \beta_{p-1}$, all the terminal states of the IOTS $(\alpha_p \,][\, Imp\ \textbf{after}(\alpha_1 \ldots \alpha_{p-1}, \beta_1 \ldots \beta_{p-1}))\downarrow_{O,\delta} \,][\, out(\alpha_p, \beta_1 \ldots \beta_{p-1})$ include all the **pass** states of $out(\alpha_i)$ and only them.
- *Imp* that is queued-suspension trace-included in the *Spec* but not queued-suspension trace-equivalent to the *Spec* is said to be *queued-suspension weakly-distinguishable* from *Spec*.

Accordingly, the following is a generalization of Proposition 10.

Proposition 15. Given IOTS *Spec* and *Imp*,

- *Imp* is queued-suspension separable from *Spec* iff there exists a finite sequence of input words $\alpha_1 \ldots \alpha_i$ such that $Qstraces_o(Imp\text{-}\textbf{\textit{after}}\text{-}(\alpha_1 \ldots \alpha_{i-1}, \gamma_1 \ldots \gamma_{i-1}), \alpha_i) \cap Qstraces_o(Spec\text{-}\textbf{\textit{after}}\text{-}(\alpha_1 \ldots \alpha_{i-1}, \gamma_1 \ldots \gamma_{i-1}), \alpha_i) = \varnothing$ for any consistent $\gamma_1 \ldots \gamma_{i-1}$.
- *Imp* that is not queued-suspension separable from *Spec* is queued-suspension distinguishable from it iff there exist a finite sequence of input words $\alpha_1 \ldots \alpha_i$ and consistent $\gamma_1 \ldots \gamma_{i-1}$ such that $Qstraces_o(Imp\text{-}\textbf{\textit{after}}\text{-}(\alpha_1 \ldots \alpha_{i-1}, \gamma_1 \ldots \gamma_{i-1}), \alpha_i) \not\subseteq Qstraces_o(Spec\text{-}\textbf{\textit{after}}\text{-}(\alpha_1 \ldots \alpha_{i-1}, \gamma_1 \ldots \gamma_{i-1}), \alpha_i)$.

- *Imp* that is not queued-suspension distinguishable from *Spec* is queued-suspension weakly-distinguishable from it iff there exist a finite sequence of input words $\alpha_1 \dots \alpha_i$ and consistent $\gamma_1 \dots \gamma_{i-1}$ such that $Qstraces_o(Imp\text{-}\mathbf{after}\text{-}(\alpha_1 \dots \alpha_{i-1}, \gamma_1 \dots \gamma_{i-1}), \alpha_i) \subset Qstraces_o(Spec\text{-}\mathbf{after}\text{-}(\alpha_1 \dots \alpha_{i-1}, \gamma_1 \dots \gamma_{i-1}), \alpha_i)$.
- *Imp* is queued-suspension trace-included into *Spec*, iff $Qstraces(t_0) \subseteq Qstraces(s_0)$.
- *Imp* and *Spec* are queued-suspension trace-equivalent iff $Qstraces(t_0) = Qstraces(s_0)$.

The queued-suspension testing also needs input and output buffers as the queued-quiescence testing. The size of the input buffer is defined by the longest input word in a chosen test case $(\alpha_1 \dots \alpha_p, Out(\alpha_1 \dots \alpha_p))$, while that of the output buffer by the longest output sequence produced in response to any input word. We assume the size of the input buffer k is given and use it to define queued-suspension k-traces and accordingly, to parameterize Definition 14 obtaining appropriate notions of k-distinguishability. In particular, a queued-suspension trace of *Spec* $\alpha_1\beta_1\delta \dots \alpha_p\beta_p\delta \in Qstraces(s_0)$ is called a queued-suspension k-trace of *Spec* if $|\alpha_i| \le k$ for all $i = 1, \dots, p$. The set of all these traces $Qstraces^k(s_0)$ has a finite representation.

Definition 16. Let S_{stable} be the set of all stable states of an IOTS $Spec = \langle S, I \cup O, \lambda, s_0 \rangle$ and I^k denote the set of all words of at most k inputs. A *queued-suspension k-machine* for *Spec* is a tuple $\langle R, I^kO^*\delta, \lambda^k_{stable}, s_0 \rangle$, denoted $Spec^k_{susp}$, where the set of states $R \subseteq \mathbf{P}(S_{stable}) \cup \{s_0\}$, ($\mathbf{P}(S_{stable})$ is a powerset of S_{stable}), and the transition relation λ^k_{stable} are the smallest sets obtained by application of the following rules:

- $(r, \alpha\beta, r') \in \lambda^k_{stable}$ if $\alpha\beta \in I^kO^*\delta$ and r' is the union of sets s-**after**-(α, β) for all $s \in r$.
- In case the initial state s_0 is unstable $(s_0, \alpha\beta, r') \in \lambda^k_{stable}$ if $\alpha\beta \in I^kO^*\delta$, $\alpha = \varepsilon$ and $r' = s_0$-**after**-(ε, β).

Notice that each system that does not oscillate has at least one stable state.

Proposition 17. The set of traces of $Spec^k_{susp}$ coincides with the set of queued-suspension k-traces of *Spec*.

Corollary 18. *Imp* is queued-suspension k-distinguishable from *Spec* iff the Imp^k_{stable} has a trace that is not a trace of $Spec^k_{susp}$.

Figure 1 gives the example of IOTS that are queued-suspension trace equivalent, recall that they are also queued-quiescent trace-equivalent, but not quiescent trace equivalent.

We notice that a queued-suspension k-machine can be viewed as an FSM with the input set I^k and output set O^m for an appropriate integer m, so that FSM-based methods could be adapted to derive queued-suspension test cases.

6. Conclusion

We addressed the problem of testing from transition systems with inputs and outputs and elaborated a testing framework based on the idea of decomposing a tester into input and output processes. Input test process is applying inputs to a system under test via a finite input queue and output test process is reading outputs that the system puts into a finite output queue until it detects no more outputs from the system, i.e., the tester detects

quiescence in the system. In such a testing architecture, input from the tester and output from the system under test may occur simultaneously. We call such a testing scenario a queued testing. We analyzed two types of queued testers, the first consisting of single input and single output test processes, a so-called queued-quiescence tester, and the second consisting of several such pairs of processes, a so-called queued-suspension tester. We defined implementation relations that can be checked in the queued testing with both types of testers and proposed test derivation procedures.

Our work differs from the previous work in several important aspects. First of all, we make a liberal assumption on the way the tester interacts with a system under test, namely that the system can issue output at any time and the tester cannot determine exactly its stimulus after which an output occurs. We believe this assumption is less restrictive than any other assumption known in the testing literature [BrTr01], [Petr01]. Testing with this assumption requires buffers between the system and tester. These buffers are finite, opposed to the case of infinite queues considered in a previous work [VTKB92]. We demonstrated that in a queued testing, the implementation relations that can be verified are coarser than those previously considered. Appropriate implementation relations were defined and test derivation procedures were elaborated with a fault model in mind. Thus, the resulting test suite becomes finite and related to the assumptions about potential faults, opposed to the approach of [Tret96], where the number of test cases is, in fact, uncontrollable and not driven by any assumption about faults. The finiteness of test cases allows us, in addition, to check equivalence relations and not only preorder relations as in, e.g., [Tret96].

Concerning future work, we believe that this paper may trigger research in various directions. One possible extension could be to consider non-rigid transition systems, allowing non-observable actions. Procedures for test derivation proposed in this paper could be improved, as our purpose here was just to demonstrate that the new testing problem with finite queues could be solved in a straightforward way. It is also interesting to see to which extent one could adapt FSM-based test derivation methods driven by fault models, as it is done in [TaPe98] with a more restrictive assumption about a tester in mind.

Acknowledgment

This work was in part supported by the NSERC grant OGP0194381. The first author acknowledges fruitful discussions with Andreas Ulrich about testing IOTS. Comments of Jia Le Huo are appreciated.

References

[BoPe94] G. v. Bochmann and A. Petrenko, Protocol Testing: Review of Methods and Relevance for Software Testing, the proceedings of the ACM International Symposium on Software Testing and Analysis, ISSTA'94, USA, 1994.

[BrTr01] E. Brinksma and J. Tretmans, Testing Transition Systems: An Annotated Bibliography, LNCS Tutorials, LNCS 2067, Modeling and Verification of Parallel Processes, edited by F. Cassez, C. Jard, B. Rozoy and M. Ryan, 2001.

[dVBF02] R. G. de Vries, A. Belinfante and J. Feenstra, Automated Testing in Practice: The Highway Tolling System, the proceedings of the IFIP 14th International Conference on Testing of Communicating Systems, TestCom'2002, Berlin, Germany, 2002.

[Glab90] R. J. van Glabbeek, The Liear Time-Branching Time Spectrum, the proceedings of CONCUR'90, LNCS 458, 1990.

[JJTV99] C. Jard, T. Jéron, L. Tanguy and C. Viho, Remote Testing Can Be as Powerful as Local Testing, the proceedings of the IFIP Joint International Conference, Methods for Protocol Engineering and Distributed Systems, FORTE XII/PSTV XIX, China, 1999.

[LyTu89] N. Lynch and M. R. Tuttle, An Introduction to Input/Output Automata, CWI Quaterly, 2(3), 1989.

[Petr01] A. Petrenko, Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography, LNCS Tutorials, LNCS 2067, Modeling and Verification of Parallel Processes, edited by F. Cassez, C. Jard, B. Rozoy and M. Ryan, 2001.

[Phal93] M. Phalippou, Executable Testers, the proceedings of the IFIP Sixth International Workshop on Protocol Test Systems, IWPTS'93, France, 1993.

[PYBD96] A. Petrenko, N. Yevtushenko, G. v. Bochmann, R. Dssouli, Testing in Context: Framework and Test Derivation, Computer Communications, 19, 1996.

[Sega93] R. Segala, Quiescence, Fairness, Testing and the Notion of Implementation, the proceedings of CONCUR'93, LNCS 715, 1993.

[TaPe98] Q. M. Tan, A. Petrenko, Test Generation for Specifications Modeled by Input/Output Automata, the proceedings of the 11th International Workshop on Testing of Communicating Systems, IWTCS'98, Russia, 1998.

[Tret96] J. Tretmans, Test Generation with Inputs, Outputs and Repetitive Quiescence, Software-Concepts and Tools, 17(3), 1996.

[Vaan91] F. Vaandrager, On the Relationship between Process Algebra and Input/Output Automata, the proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science, 1991.

[VTKB92] L. Verhaard, J. Tretmans, P. Kim, and E. Brinksma, On Asynchronous Testing, the proceedings of the IFIP 5th International Workshop on Protocol Test Systems, IWPTS'92, Canada, 1992.

Optimization Problems in Testing Observable Probabilistic Finite-State Machines

Fan Zhang[1] and To-yat Cheung[2]

[1]Department of Computing, Hong Kong Polytechnic University
e-mail: csfzhang@comp.polyu.edu.hk

[2]Department of Computer Science, City University of Hong Kong
e-mail: cscheung@cityu.edu.hk

Abstract: In this paper, we investigate the transfer sequence and diagnosis sequence for testing a probabilistic finite-state machine whose transitions have weights. These sequences are adaptive and are measured by their average length. We discuss the problem of optimal strategies for selecting input and thus for generating these sequences. In particular, we show that, if the probabilistic machine is *observable* (i.e., the next-state of each transition can be uniquely determined by its output), then, polynomial-time algorithms can be obtained for the following problems: (1) Find the shortest transfer sequence from a start state to a target state, or prove that no such a sequence exists. (2) Find the shortest pair-wise distinguishing sequences for a given pair of states, or prove that no such sequence exists.

Keywords: Average weight, Diagnosis tree, Probabilistic finite-state machine, Software testing, Transfer tree.

I. INTRODUCTION

Nondeterminism is common in testing systems that have concurrent processes and internal actions. Recently, research interest in testing has been extended from *deterministic finite-state machines* (DFSM) [1,4,8,13] to *nondeterministic finite-state machines* (NFSM) [2,3,5,7,9,12] and *probabilistic finite-state machines* (PM) [2,3,14,15]. A PM is an NFSM with transition probabilities that represent the chance of a transition to be triggered by the input.

In the state-based approach of testing, the system under test is specified as an finite-state machine M and a test sequence is designed based on M. During testing, the input portion of the sequence is applied to the *implementation under test* (IUT) and the actual outputs are compared with the expected ones to uncover any possible output or state-transition errors. Two major test activities must be carried out in this approach: *state-transfer* and s*tate-identification*. For state-transfer, the problem is to find a sequence of

inputs that drives the IUT from its initial state *s* to a targeted state *t*. For a DFSM, the inputs along any directed path connecting *s* to *t* will serve this purpose. For an NFSM, however, this problem becomes much more complex because the same input sequence may bring the IUT sometimes to *t* and sometimes elsewhere. Therefore, instead of a single path, we have to find a *transfer-tree* (TT) of which every path terminates at *t*, so as to always bring the IUT to the expected targeted state.

State-identification is to verify the identification of a state. During testing, the current state of the IUT cannot be observed directly but it can only be inferred from the output observed. After the IUT is brought to a state *t'* that is expected to be *t*, the identification of *t'* need to be checked to make sure the reached state *t'* is *t* indeed. For DFSM, a diagnosis sequence of input/output pairs for *t* is derived. The following three kinds of diagnosis sequences have been used most frequently: A *distinguishing sequence* can identify the current state among all possible states. That is, the same distinguishing sequence can be used for every state. A *UIO sequence* can differentiate a specific state *r* from all other states. Different UIO sequences may be needed for different specified states [11]. A *pair-wise distinguishing sequence* can identify the current state between two given states. For an NFSM, similar to generalizing a transfer sequence to a TT, a diagnosis sequence is generalized to a diagnosis tree (DT). For a general NFSM or PM, most of the problems are computationally difficult. For example, deciding the existence of a TT or DT is Exptime-complete; also, the algorithms for finding TTs and DTs require exponential time [2].

For testing purpose, we believe that the PM is a better model than the NFSM. The main reason is that the test sequence for a PM can be measured but cannot be (at least not fairly) for an NFSM. Another reason is that the PM can reflect the stochastic behavior pattern of the system under test but the NFSM cannot.

This paper reports our recent investigation on TTs and DTs for an *observable* PM (OPM), a special but important class of PM [10]. In such a machine, the next-state of each transition can be uniquely determined by its output. The transitions of the PM have weight representing cost or time. The test sequences can be measured by their average length. We discuss the problem of optimal strategies for selecting input and thus for generating test sequences. In particular, we show that for an OPM, polynomial-time algorithms can be obtained for the following problems: (1) Find the shortest transfer sequence from a start state to a target state, or prove that no such a sequence exists

(Section III). (2) Find the shortest pair-wise distinguishing sequences for a given pair of states, or prove that no such sequence exists (Section IV).

II. DEFINITIONS

This section gives definitions. Most of them can be found in [15].

Definition 2.1 A *nondeterministic finite-state machine* M is a quintuple (S, I, O, D, λ), where S is a finite set of *states*, I is a finite set of inputs, O is a finite set of outputs, $D \subseteq S \times I$ is the domain of the state-input pairs, and λ is a mapping from D to $O \times S$. A *transition* (*i*,*j*;*a*/*y*) starts at state *i*, ends at state *j* and is associated with an input/output pair *a*/*y*. The *symbol* ε denotes "no input" or "no output".

- M is called a *probabilistic machine* (PM) if it is associated with a probability function ρ: $S \times S \times I \times O \rightarrow [0,1]$ such that $\rho(i,j;a/y) > 0$ if and only if (*i*,*j*;*a*/*y*) is a transition and that $\sum_{y,j} \rho(i,j;a/y) = 1$ for every $(i,a) \in D$.
- M is further said to be an *observable PM* (*OPM*) if, for any two transitions (i,j_1; a/y_1) and (i,j_2; a/y_2) with the same (*i*,*a*), $y_1 = y_2$ implies $j_1 = j_2$.
- For the rest of the paper, M always represents an OPM with a *weight function* ω: $S \times S \times I \times O \rightarrow [0,\infty)$ such that $\omega(i,j;a/y) > 0$ for every transition (*i*,*j*;*a*/*y*).

Example 1. Figure 1 shows an OPM N represented by a directed graph. In this OPM, every transition has weight 1 and probability either 1 or 0.5 (not shown), depending on whether there are one or two outgoing transitions with the same input. N is strongly connected, i.e., for any two states *s* and *s'*, there is a (directed) path from *s* to *s'*.

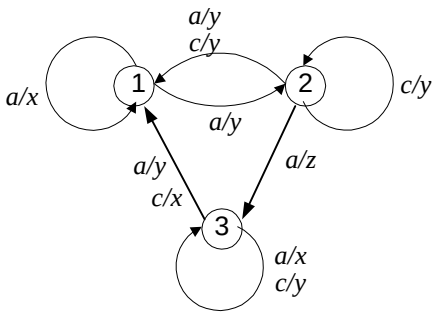


Figure 1. An OPM N.

Definition 2.2 A *single-input tree T* for M is formed by all possible execution paths under an input-select strategy *g*. It is defined as follows: (1) *T* contains the starting state *s* as its root. (2) At any node *v* of *T* whose corresponding state is *i*, there are two cases: *Case 1*. No input is selected (as a testing goal has been achieved), *v* is a terminal node and has label (*i*, ε). *Case 2*. An input *a* is selected by *g*, and the node in *T* has label (*i*, *a*);

also, for every outgoing transition ($i,j;a/y$) from i, an arc with label y and ending node j will be attached to the node in T. (Note that a single-input tree may be infinite.)

Definition 2.3 Let T be a single-input tree for M and P be a path of T.

- $\rho(P)$, the *probability of P*, is the product of the probabilities of its arcs (transitions).
- $\omega(P)$, the *weight of P*, is the sum of the weights of its arcs.
- P is called a *full path* if it starts from the root and ends at a terminal node of T.
- $\rho(T)$, the *probability* of reaching a terminal node of T is defined as $\text{limit}_{m\to\infty}\sum(\rho(P)$: for all full paths P of T with length $\le m$) and is denoted by $\sum_{P\subseteq T}\rho(P)$.
- T is said to be *almost sure* if $\rho(T) = 1$.
- When T is almost sure, its *average weight* $\omega(T)$ is defined as $\sum_{P\subseteq T}\rho(P)\omega(P)$. $\omega(T)$ measures the efficiency of reaching a terminal node of T from its root.
- T is said to be a t-targeted *transfer tree* (TT) at state s if it is an almost sure single-input tree whose root is s and terminal nodes all have label (t,ε).
- A *policy* f of M selects an input for each state, that is, $f: S \to I \cup \{\varepsilon\}$.
- T_s^f denotes the single-input tree at s *determined by* f, whose labels are $(i,f(i))$.

Example 2. Figure 2 shows the top part of a t-targeted TT ($t = 1$) at state 3 for OPM N (Figure 1): T_3 is determined by policy f: $f(1) = \varepsilon$, $f(2) = c$ and $f(3) = c$. The probability and weight of every arc of T_2 are 1/2 and 1, respectively. T_3 is almost sure, as $\rho(T_3) = 1/2 + (1/2)^2 + ... + (1/2)^k + ... = 1$. It average weight is $\omega(T_3) = 1(1/2) + 2(1/2)^2 + 3(1/2)^3 + ... + k(1/2)^k + ... = 2$.

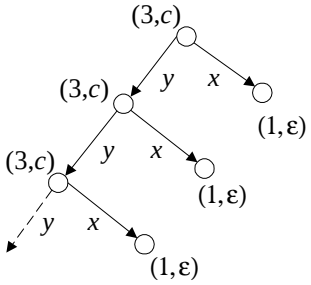


Figure 2. A t-targeted TT T_3 for OPM N.

Definition 2.4 Let M be an OPM with target t and f be a policy for M.

- M^f denotes the submachine *induced* on M by f. That is, $M^f = (S, I, O, D^f, \lambda)$, where D^f is the domain $\{(i,f(i)): i \in S\}$.
- M is said to be *t-targeted* if for every state v of M there is a directed path from v to t.
- f is a *t-policy* if M^f is t-targeted.

We mention two useful facts about *t*-policies [15]: (a) If f is a *t*-policy, then T_i^f is a *t*-targeted TT for every $i \in S$. (b) If M is *t*-targeted, then there is a polynomial-time algorithm for finding a *t*-policy f for M.

III. OPTIMAL TRANSFER-TREES

Now we consider the problem of finding an optimal policy for M that generates TTs having *minimum average weight*. We first consider the case where M has a *t*-policy.

Definition 3.1 Let f be a policy for M = (S, I, O, D, λ), $n = |S|$, $i, j \in S$ and $a \in I \cup \{\varepsilon\}$.

- $\mu(i,a,j)$ denotes the total probability of all the transitions which start at state i, have input a and end at state j, i.e., $\mu(i,a,j) = \sum_y \rho(i,j;a/y)$ and $\mu(i,\varepsilon,j) = 0$.
- A^f denotes the $n \times n$ matrix (a_{ij}), where $a_{ij} = \mu(i,f(i),j)$.
- E denotes the $n \times n$ identity matrix.
- b_{ia} denotes the average weight of those transitions outgoing from i and activated by input a, i.e., $b_{ia} = \sum_{y,j} \rho(i,j;a/y)\omega(i,j;a/y)$ and $b_{i\varepsilon} = 0$.
- $\mathbf{b}^f$ denotes the vector $(b_i: i \in S)$, where $b_i = b_{if(i)}$.

Optimality Condition of a *t*-policy f [15]: f is optimal if and only if $\mathbf{z} = (E - A^f)^{-1}\mathbf{b}^f$ satisfies the following inequalities:

$$z_i \le b_{ia} + \sum_{j \in S} \mu(i,a,j)z_j, \qquad \text{for every } (i,a) \in D \text{ and } i \ne t.$$

Algorithm Optimal-Policy

Given: A *t*-targeted OPM M = (S, I, O, D, λ).

Result: An optimal policy h for generating the shortest transfer trees.

Step 1: Find an optimal solution $\mathbf{z}^*$ to the following linear program (LP):

$$\begin{aligned} \text{LP:} \quad & \text{maximize} && \sum(z_j: j \in S) \\ & \text{Subject to:} && z_i - \sum_{j \in S} \mu(i,a,j)z_j \le b_{ia}, \text{ for every } (i,a) \in D \\ & && z_t = 0 \end{aligned}$$

Step 2: Let M* be the OPM induced by $D^* = \{(i,a) \in D: z^*_i - \sum_j \mu(i,a,j)z^*_j = b_{ia}\}$. Find a *t*-policy h of M*.

Theorem 1: Algorithm Optimal-Policy is correct and has polynomial-time complexity.

Sketch Proof: Consider the dual linear program of the LP:

$$\begin{aligned} \text{DLP:} \quad & \text{minimize} && \sum(b_{ia}x_{ia}: (i,a) \in D) \\ & \text{Subject to} && \sum_{a \in I} x_{ja} - \sum_{(i,a) \in D} x_{ia}\mu(i,a,j) = 1, \ j \in S\backslash\{t\} \\ & && x_{ia} \ge 0, \quad (i,a) \in D \end{aligned}$$

As M has a *t*-policy, say f, we construct **x*** based on this f as follows: For all $(i,a) \in D$, $x^*_{ia} = x^*_{if(i)}$ if $a = f(i)$; $x^*_{ia} = 0$ if $a \neq f(i)$; and $x^*_{i\varepsilon} = 0$. **x*** satisfies the equations of the DLP and especially,

$$x_{jf(j)} - \sum_i \mu(i,f(i),j)x_{if(i)} = 1, \text{ for all } j \in S\backslash\{t\}, \text{ and } x_{tf(t)} = 1,$$

which is, in matrix form,

$$(x_{if(i)}: i \in S)(E - A^f) = (1,1, \ldots,1).$$

Furthermore, as $(E - A^f)^{-1}$ exists and is non-negative, $(x^*_{if(i)}: i \in S) = (1,1,\ldots,1)(E - A^f)^{-1}$ is non-negative. Hence, **x*** is a feasible solution to the DLP, and the LP has a finite solution **z***.

It can be easily shown that M* has a *t*-policy h. As all $(i, h(i)) \in D^*$, **z*** satisfies $z^*_i - \sum_j \mu(i,h(i),j)z^*_j = b_{ih(i)}$, and hence $\mathbf{z}^* = (E - A^h)^{-1}b^h$. Because **z*** satisfies the Optimality Condition, h is an optimal *t*-policy.

The LP can be solved in polynomial time [6], yielding a finite optimal solution **z***. As mentioned in Section II, a *t*-policy for M can be obtained in polynomial-time. ∎

For an arbitrary OPM M, one can reduce M into an OPM M' that has a *t*-policy [15] and then apply the algorithm presented here.

IV. DIAGNOSIS TREES

This section investigates the problem of finding a pair-wise distinguishing tree that has a minimum average weight for an OPM, and also discusses UIO trees and distinguishing trees.

Definition 4.1 For two states *t* and *r* of an OPM M, a *pair-wise distinguishing tree* (PDT) *T* is an almost-sure single-input tree rooted at (*t*, *r*) that can decide the state under test to be either *t* or *r* whenever the input/output sequence of any full path of *T* is executed.

An example of such a PDT for OPM N (Figure 3a) is given in Figure 3b. The symbol "~" stands for "state does not exist".

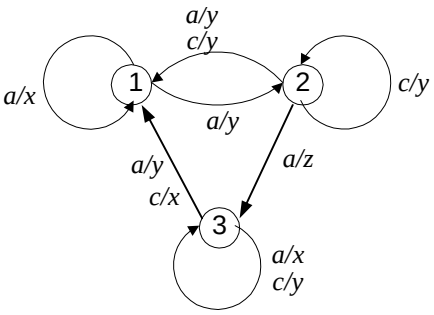


Figure 3a. An OPM N

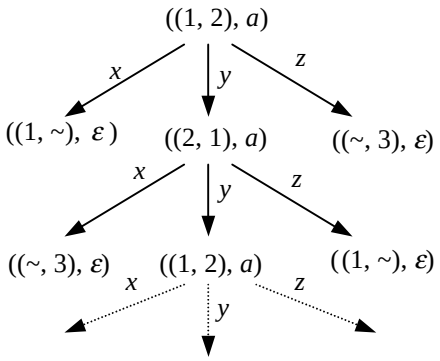


Figure 3b. A PDT for (1, 2).

The problem of creating a general PDT was first investigated by Alur et al [2]. We shall find the minimum-length PDT by a similar approach, that is, to transform the problem of finding a PDT to a problem of finding a TT. It can be briefly described as follows: First, construct an auxiliary ONFSM M' from M. The state set of M' is a subset of $(S\times S) \cup\{\theta\}$, where S is the state set of M and θ is the target state representing the cases where an inference of the state identification can be drawn. Next, find an optimal TT for M' from the start state (t, r) to the targeted state θ. One can show that an optimal TT for M' with weight and probability assigned according to the algorithm below is essentially an optimal PDT for M. For the following algorithm, we assume that M is completely specified, that is, $D = S \times I$.

Algorithm Optimal-PDT

Given: An OPM $M = (S, I, O, D, \lambda)$ and two states $t, r \in S$, where the probability of the state being t is p and the probability being r is $q = 1 - p$.

Result: An OPM $M' = (S', I', O', D', \lambda')$ and an optimal θ-policy f for M'. If $f(t, r) \neq \varepsilon$, then an Optimal PDT can be obtained by applying f on M'. Otherwise, PDT for (t, r) does not exist.

Method:

Step 1. Construct M' from M as follows, where ρ' and ω' are the probability and weight functions of M', respectively:

$S' \leftarrow (S \times S) \cup \{\theta\}$, $I' \leftarrow I$, $O' \leftarrow O$, $D' \leftarrow (S' \setminus\{(v,v): v \in S\}) \times I$;

for (every $(i,j) \in S'\setminus\{\theta\}$ such that $i \neq j$ and every input a)

for *every* transition $\boldsymbol{e}_1 = (i, i'; a/y)$, construct a transition $\boldsymbol{e}$ of M' as follows:

If there exists j' such that $\boldsymbol{e}_2 = (j, j'; a/y)$ is a transition of M,

$\boldsymbol{e} = ((i,j), (i',j'); a/y)$; /* add $((i',j'), y)$ to $\lambda'((i,j), a)$ */

$\rho'(\boldsymbol{e}) = p\rho(\boldsymbol{e}_1) + q\rho(\boldsymbol{e}_2)$, $\omega'(\boldsymbol{e}) = p\omega(\boldsymbol{e}_1) + q\omega(\boldsymbol{e}_2)$

else (no such j' exists)

$\boldsymbol{e} = ((i,j), \theta; a/y)$. /* add (θ, y) to $\lambda'((i,j), a)$ */

$\rho'(\boldsymbol{e}) = p\rho(i, i'; a/y)$, $\omega'(\boldsymbol{e}) = p\omega(\boldsymbol{e}_1)$;

end for

for every transition $\boldsymbol{e}_2 = (j, j'; a/y)$ such that no transition $(i, i'; a/y)$ exists, construct a transition $\boldsymbol{e}$ of M' as follows:

$\boldsymbol{e} = ((i,j), \theta; a/y)$. /* add (θ, y) to $\lambda'((i,j), a)$ */

$\rho'(\boldsymbol{e}) = q\rho(\boldsymbol{e}_2)$, $\omega'(\boldsymbol{e}) = q\omega(\boldsymbol{e}_2)$;

end for

end for

Step 2. Find an optimal policy f for M' with targeted state θ by applying Algorithm Optimal-Policy of Section III.

Theorem 2 Algorithm Optimal-PDT finds a policy f for generating a PDT with minimum average weight (when $f(t, r) \neq \varepsilon$) or concludes that no PDT exists for the pair (t, r) (when $f(t, r) = \varepsilon$).

Remark on the Algorithm Optimal-PDT:

(a) If $p = 1$ and $q = 0$, then the policy f is to verify the current state is t but not r.

(b) If $p = 0$ and $q = 1$, then the policy f can be used to verify the current state is r but not t.

(c) If both p and q are not 0, then the policy f can identify the state as either t or r.

(d) The way of assigning the transition probability and weight of M' ensures that M' is an OPM and that the PDT is optimal.

Definition 4.2: For a given state s of M, a *unique input output tree* (UIOT) is an almost sure single-input tree T of M rooted at s such that, when the input sequence of any full path P of T is applied to a different state, the output sequence is always different from that of P.

One of the approaches for obtaining the shortest UIOT for a state s is to look into the state space S^n, using ideas similar to the PDT. A UIOT for state 1 of OPM N is shown in Figure 4a in the space S^n, where T_1 is a PDT ($p = 1$, $q = 0$) for the pair (1, 3) and T_2 is a PDT ($p = 1$, $q = 0$) for the pair (2, 1). For the arc from ((1,2,3), a) to ((1,~,3), a) of the UIOT, its probability and weight are respectively assigned 0.5 and 1, same as those of the transition (1, 1; a/x) of N. For the arc from ((1,2,3), a) to ((2,1,1), a), its probability and

weight are respectively assigned those of the transition (1, 2; *a/y*). In general, the probabilities and weights of a UIOT for state *s* are the same as those of the corresponding single-input tree starting at *s*.

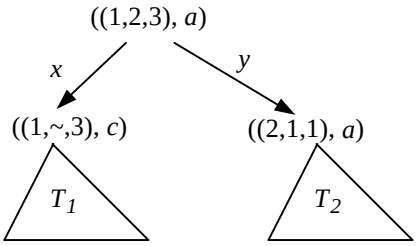


Figure 4a. A UIOT for state 1

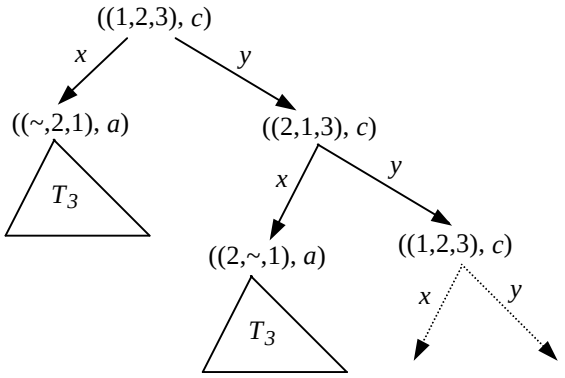


Figure 4b. A distinguishing tree for OPM N

Definition 4.3: A *distinguishing tree* for M is an almost sure single-input tree *T* such that, whenever the end of a full path of *T* is reached, the initial state is uniquely identified.

As an example, Figure 4b shows a distinguishing tree for OPM N (Figure 3a), where T_3 is a PDT for the pair (2, 1). As for a PDT, the average weight of a distinguishing tree is related to the probabilities $p_1, \ldots, p_n$, where p_i represents the probability of the state under test being state *i*.

V CONCLUDING REMARKS

In this paper, we described a polynomial-time algorithm for finding the shortest transfer tree for an observable probabilistic finite-state machine. We also reduced the problem of finding the shortest pair-wise distinguishing tree into the shortest transfer-tree problem.

Many problems remain unsolved and require further research. For example, for state transfer, a direct combinatorial and polynomial-time algorithm for finding an optimal policy is desirable. For state identification, algorithms for generating the optimal UIOT and DT need to be investigated.

ACKNOWLEDGEMENT

We thank the referees for their comments. This work is supported by the Hong Kong Polytechnic University Research Grant G-YC51.

REFERENCES

1. Aho, A.V., Dahbura, A.T., Lee, D. and Uyar, M.U. (1991), "An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours," *IEEE Trans. on Commun.*, 39 (11), pp. 1604-1615.
2. Alur, R., Courcoubetis, C. and Yannakakis, M. (1995), "Distinguishing tests for nondeterministic and probabilistic machines", *Proc. Twenty-Seventh Annual ACM Symposium on Theory of Computing*, Las Vegas, Nevada, pp. 363-372.
3. Cheung, T. and Ye, X. (1995), "A fault-detection approach to the conformance testing of nondeterministic systems", *Journal of Parallel and Distributed Computing* 28, pp. 94-100.
4. Chow, T.S. (1989), "Testing software design modeled by finite-state machines", *IEEE Trans. on Software Eng.* 4 (3), pp. 178-187.
5. Fujiwara, S. and Bochmann, G.v. (1992), "Testing non-deterministic state machines with fault coverage", *Protocol Test Systems*, IV, (J. Kroon, R.J. Heijink, and E. Brinksma eds.), pp. 267-280.
6. Khachiyan, L.G. (1979), "A polynomial algorithm in linear programming", *Soviet Math Dolk.* 20, pp. 191-194.
7. Kloomsterman, H. (1993), "Test derivation from nondeterministic finite-state machines", *Protocol Test Systems* V, (Bochmann, G.v., Dssouli, R. and Das, A. Eds.), North Holland, pp. 297-308.
8. Lee, D. and Yannakakis, M. (1996), "Principles and methods of testing finite state machines - a survey", *Proc. IEEE* 84, pp. 1090-1126.
9. Low, S. (1993), "Probabilistic conformance testing of protocols with unobservable transitions", *Proc. Intl. Conf. on Network Protocols*, pp. 368-375.
10. Luo, G., Bochmann, G.v., Das, A. and Wu, C. (1992), "Failure-equivalent transformation of transition systems to avoid internal actions", *Infor. Processing Letters* 44, pp. 333-343.
11. Sabnani, K. and Dahbura, A. (1988), "A protocol test generation procedure", *Computer Networks and ISDN Systems* 15, pp. 285-297.
12. Tripathy, P. and Naik, K. (1993), "Generation of adaptive test cases from nondeterministic finite state models", *Protocol Test Systems* V, (Bochmann, G.v., Dssouli, R. and Das, A. Eds.), North Holland, pp. 309-320.
13. Ural, H., Wu, X. and Zhang, F. (1997), "On minimizing the lengths of checking sequences", *IEEE Trans. on Computers* 46 (1), pp. 93-99.
14. Yi, W. and Larsen, K.G. (1992), "Testing probabilistic and nondeterministic processes", *Protocol Specification, Testing, and Verification* XII, pp. 47-62.
15. Zhang, F and Cheung T.Y. (2002) "Optimal transfer trees and distinguishing trees for testing observable nondeterminstic finite-state machines", *IEEE Trans on Soft Eng*, accepted.

BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming

F. Ambert, F. Bouquet, S. Chemin, S. Guenaud,
B. Legeard, F. Peureux, N. Vacelet
Université de Franche-Comté, France

M. Utting
The University of Waikato, New Zealand

Abstract

In this paper, we present an environment for boundary-value test generation from Z and B specifications. The test generation method is original and was designed on the basis of several industrial case-studies in the domain of critical software (Smart Card and transport areas). It is fully supported by a tool-set: the BZ-Testing-Tools environment. The method and tools are based on a novel, set-oriented, constraint logic programming technology. This paper focusses on how this technology is used within the BZ-TT environment, how Z and B specifications are translated into constraints, and how the constraint solver is used to calculate boundary values and to search for sequences of operations during test generation.

Key words: Computer-Aided Software Testing Tool, Specification-based test generation, boundary value testing, B notation, Z notation

1 Introduction

From the end of the 1980's, specification-based test generation has been a very active and productive research area. In particular, formal specification has been clearly recognized as a very powerful input for generating test data [BGM91, DF93, Tre96, FJJV96], as well as for oracle synthesis [RO92].

In [LPU02b], we presented a new approach for test generation from set-oriented, model-based specifications: the BZ-TT method. This method is based on constraint logic programming (CLP) techniques. The goal is to test every operation of the system at every *boundary state* using all *input boundary values* of that operation. The unique features of the BZ-TT method are that it:

- takes both B [Abr96] and Z [Spi92] specifications as input;
- avoids the construction of a complete finite state automaton (FSA) for the system;
- produces boundary-value test cases (both boundary states and boundary input values);
- produces both negative and positive test cases;

- is fully supported by tools;
- has been validated in several industry case studies (GSM 11-11 smart card software [LP01, BLLP02], Java Card Virtual Machine Transaction mechanism [BJLP02], and a ticket validation algorithm in the transport industry [CGLP01]). The method is currently being used in another industrial project to generate tests for an automobile windscreen wiper controller.

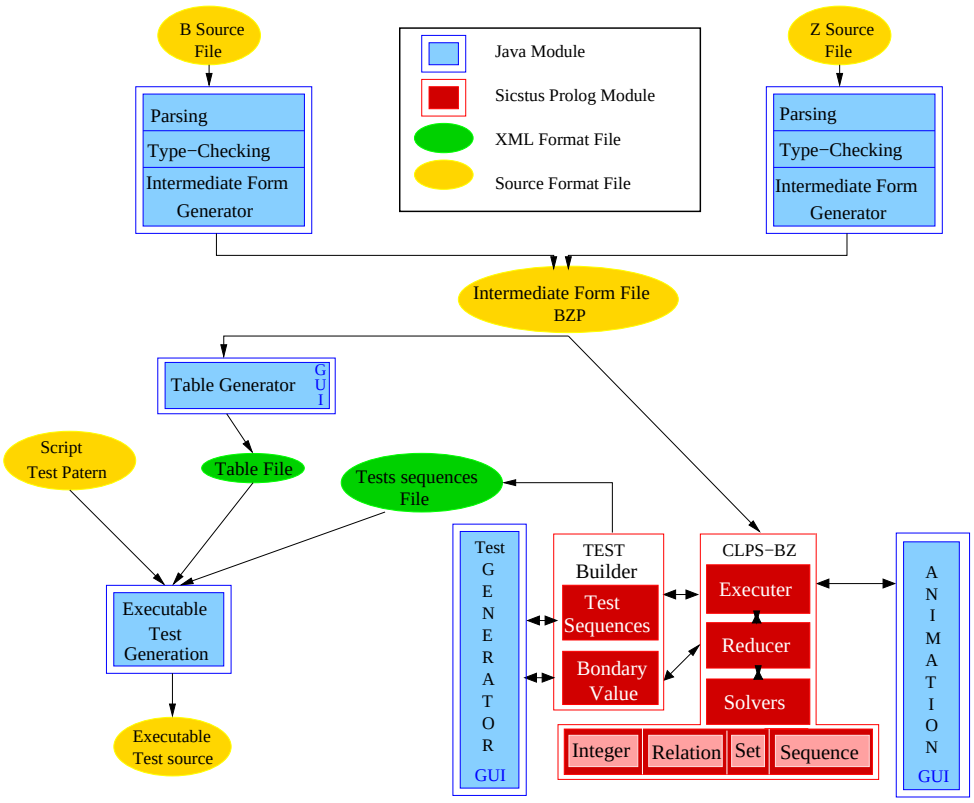


Figure 1: Overall Architecture of the BZ-TT Environment

Figure 1 shows the architecture of the BZ-TT environment. The key component in this environment is the CLPS-BZ solver, which is used to calculate boundaries and simulate the execution of operations. It is this solver which enables the test generation process to be effectively automated. Furthermore, since it is a general reasoning engine, whose input is a pre/post specification format, it could potentially be used with notations other than Z or B, and for purposes other than test generation.

This paper describes how the CLPS-BZ solver is used to support the BZ-TT method of test generation and how it enables both Z and B specifications to be represented as constraint systems. Section 2 gives an overview of the BZ-TT test generation method. Section 3 describes how Z and B specifications are mapped into a common format: the BZP intermediate form. Section 4 describes the CLPS-BZ constraint solver, and how BZP predicates are represented as sets of constraints. Section 5 describes how the test generation modules use the CLPS-BZ solver. Section 6 discusses related work; Section 7 gives conclusions and outlines future work.

2 Overview of the BZ-TT test generation method

Our goal is to test some implementation, which is not derived via refinement from the formal model. The implementation is usually a state machine with hidden state. We specify this state machine by a B or Z formal specification, which has a state space (consisting of several state variables) and a number of operations that modify this state.

A behavior of such a system can be described in terms of a sequence of operations (a trace) where the first is activated from the initial state of the machine. However, if the precondition of an operation is false, the effect of the operation is unknown, and any subsequent operations are of no interest, since it is impossible to determine the state of the machine. Thus, we define a positive test case to be any legal trace, i.e. any trace where all preconditions are true. A positive test case corresponds to a sequence of system states presenting the value of each state variable after each operation invocation. The submission of a legal trace is a success if all the output values returned by the concrete implementation during the trace are equivalent (through a function of abstraction) to the output values returned by its specification during the simulation of the same trace (or included in the set of possible values if the specification is non-deterministic). A negative test case is defined as a legal trace plus a final operation whose precondition is false. The generation of negative test cases is useful for robustness testing.

The BZ-TT method consists of testing the system when it is in a *boundary state*, which is a state where at least one state variable has a value at an extremum – minimum or maximum – of its sub-domains. At this boundary state, we want to test all the possible behaviors of the specification. That is, the goal is to invoke each *update operation* with extremum values of the sub-domains of the input parameters. The test engineer partitions the operations into *update operations*, which may modify the system state, and *observation operations*, which may not.

We divide the trace constituting the test case into four subsequences:[1]

Preamble: this takes the system from its initial state to a boundary state.

Body: this invokes one update operation with input boundary values.

Identification: this is a sequence of observation operations to enable a pass/fail verdict to be assigned.

Postamble: this takes the system back to the boundary state, or to an initial state. This enables test cases to be concatenated.

The body part is the critical test invocation of the test case. Update operations are used in the preamble, body and postamble, and observation operations in the identification part.

The BZ-TT generation method is defined by the following algorithm, where $\{bound_1, bound_2, ..., bound_n\}$ and $\{op_1, op_2, ..., op_m\}$ respectively define the set of all boundary states and the set of all the update operations of the specification:

[1] The vocabulary follows the ISO9646 standard [ISO].

```
for i←1 to n                  % for each boundary state
   preamble(bound_i);         % reach the boundary state
   for j←1 to m               % for each update operation
      body(op_j);             % test op_j
      identification;         % observe the state
      postamble(bound_i);     % return to the boundary state
   endfor
   postamble(init);           % return to the initial state
endfor
```

This algorithm computes positive test cases with valid boundary input values at body invocations. A set of one or more test cases, concatenated together, defines a test sequence. For negative test cases, the body part is generated with invalid input boundary values, and no identification or postamble parts are generated, because the system arrives at an indeterminate state from the formal model point of view. Instead, the test engineer must manually define an oracle for negative test cases (typically something like *the system terminates without crashing*).

After positive and negative test cases are generated by this procedure, they are automatically translated into executable test scripts, using a test script pattern and a reification relation between the abstract and concrete operation names, inputs and outputs.

3 The BZP intermediate form

B and Z have many similarities. They both support model-oriented specification and they have similar operator toolkits and type systems. The main difference between them is that in Z the *schema calculus* is used to structure specifications and specify states and operations in a flexible way, whereas B provides an abstract programming notation (the *generalized substitution language*) for specifying operations, plus a specialized *machine* construct for specifying hierarchies of state machines.

The BZ-TT environment supports both B and Z specifications by translating them into a common notation, called BZP (B/Z Prolog format). Since the underlying CLPS-BZ solver manipulates constraints, which are restricted kinds of predicates, operations are specified in BZP using pre/post predicates.

Figure 2 gives an overview of the translation process. The generation and discharge of well-formedness proofs for B is done using standard B tools (typically before test generation begins). The B to BZP translator is written in Java, and uses the rules from the B book [Abr96, Chap. 6] to translate the generalized substitution language into pre/post predicates. The first stage of the Z to BZP translator uses standard Z tools, while the second stage is a specific translator.

For simplicity, and to improve the scalability of the test generation process, we impose three restrictions on the input specification. Firstly, it must specify a single machine. For B, this means that we allow only one abstract machine, without layering etc. For Z, we must identify the state, initialization and operation schemas of the machine. Secondly, operations must have explicit preconditions. For Z this means preconditions must be calculated (either manually, or using a theorem prover) before translation. This is good engineering practice anyway. In B, operations usually have explicit preconditions, but it is possible for the precondition to be distributed throughout the operation. We require the entire precondition to appear at the beginning of the operation, and also require this precondition to be strong

enough to ensure that the operation is feasible. Third, all the data structures must be finite, which means that the given sets are either enumerated or of a known finite cardinality. If everything is finite, then all specifications are executable (by labelling if necessary).

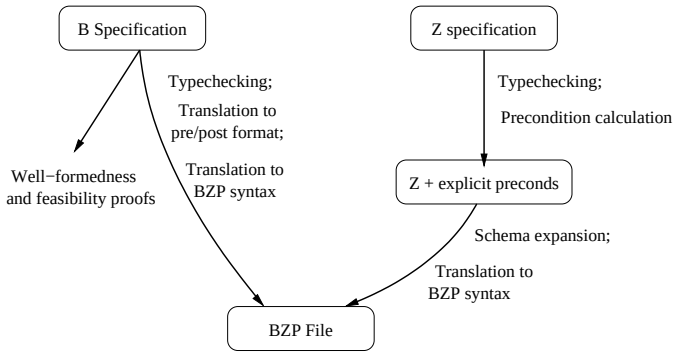


Figure 2: Translation of B and Z specifications into common BZP format.

The BZP format is a Prolog-readable syntax which supports the union of the B and Z toolkits and provides special constructs for defining state machines and operations. The syntax of expressions and predicates is similar to the Atelier-B ASCII notation for B (`*.mch` syntax). A BZP specification contains an unordered set of facts, each of which has one of the following forms:

specification(*Spec_Name*).
predicat(*Spec_Name*, *PREDICAT_KIND*, *ID*, *Pred*).
declaration(*Spec_Name*, *DECL_KIND*, *Name*, *Type*).
operation(*Spec_Name*, *Operation_Name*).

The *ID* terms are used only for reporting errors, and are typically the line number where that predicate appeared in the original specification source.

The *PREDICAT_KIND* and *DECL_KIND* terms relate each fact to some section of the original B machine (and similar constructs in Z), and are defined as follows. *PREDICATE_KIND* indicates the role of the predicate, and must be one of the constants: `constraint`, `property`, `invariant`, `initialisation`, `assertion` or one of the terms `pre(`*OpName*`)` or `post(`*OpName*`)`.

Similarly, *DECL_KIND* indicates what kind of name is being declared, and must be one of the constants: `parameter`, `set`, `constant`, `variable`, `definition` or one of the terms `input(`*OpName*`)` or `output(`*OpName*`)`. For `definition`, the type is a singleton set containing the right-hand-side of the definition.

Invariants have different roles in B and Z. This difference is visible in the resulting BZP file, since when we translate a Z operation, we include the (primed) invariant in its postcondition because it often contributes to the effect of the operation, but when we translate a B operation we do not include the invariant. However, the well-formedness proofs for a B machine check that every operation preserves the invariant, so discharging these proofs ensures that the B and Z approaches are equivalent.

To illustrate how Z specifications are translated to BZP format, we translate the following fragment of a simple process scheduler [DF93]

$$PID ::= p1 \mid p2 \mid p3 \mid p4$$

Scheduler

$active, ready, waiting : \mathbb{F}\ PID$

$\#active \leq 1$
$ready \cap waiting = \emptyset$
$active \cap waiting = \emptyset$
$active \cap ready = \emptyset$
$(active = \emptyset) \Rightarrow (ready = \emptyset)$

New

$\Delta Scheduler$
$pp? : PID$

$pp? \notin active \cup ready \cup waiting$
$waiting' = waiting \cup \{pp?\}$
$ready' = ready$
$active' = active$

The declaration of *PID* and the state schema produce:

```
machine(scheduler).

declaration(scheduler, set, pid, {p1, p2, p3, p4}).
declaration(scheduler, variable, ready, power(pid)).
declaration(scheduler, variable, active, power(pid)).
declaration(scheduler, variable, waiting, power(pid)).

predicat(scheduler, invariant, 3 , card(active) =< 1).
predicat(scheduler, invariant, 4 , ready /\ waiting = {}).
predicat(scheduler, invariant, 5 , active /\ waiting = {}).
predicat(scheduler, invariant, 6 , active /\ ready = {}).
predicat(scheduler, invariant, 7 , (active =  {}) => (ready ={})).
```

The *New* operation produces:

```
operation(scheduler, new).
declaration(scheduler,input(new), pp, pid).
predicat(scheduler, pre(new), 10 , pp /: active \/ ready \/ waiting).
predicat(scheduler, post(new), 11 , prime(waiting) = waiting \/ {pp} ).
predicat(scheduler, post(new), 12 , prime(ready) = ready).
predicat(scheduler, post(new), 13 , prime(active) = active).
predicat(scheduler, post(new), 3 , card(prime(active)) =< 1).
predicat(scheduler, post(new), 4 , prime(ready) /\ prime(waiting) = {}).
predicat(scheduler, post(new), 5 , prime(active) /\ prime(waiting) = {}).
predicat(scheduler, post(new), 6 , prime(active) /\ prime(ready) = {}).
predicat(scheduler, post(new), 7 , (prime(active) =  {}) => (prime(ready) ={})).
```

4 The CLPS-BZ solver

The CLPS-BZ solver is implemented in SICStus Prolog, and is comprised of three subsystems:

The Executer: which manages the execution of the specified machine. The Executer obtains predicates from the BZP specification, and passes them to the other modules to achieve the effect of executing the desired operation.

The Reducer: which translates BZP predicates into lower-level constraints.

The Constraint Store: which records the current state of the specified machine after some sequence of operations. In fact, we use symbolic execution, so a single constraint store typically corresponds to many possible concrete states of the specified machine.

We now describe these three subsystems, in a bottom-up fashion.

4.1 The Constraint Store

The CLPS-BZ solver manages a *constraint store*, which encodes a predicate (or a set of states) over the variables of the formal model. The constraint store contains both set-oriented constraints and finite domain (integer) constraints. The former are managed by a customized set-constraint solver called CLPS, while the latter are managed by the SICStus Prolog *CLP(FD)* solver. In this section we focus only on the set part of the store (CLPS).

The CLPS solver was originally developed for solving combinatorial problems, and is designed to work on homogeneous hereditary finite sets [ALL96]. It provides five primitive constraint relations, which are the classical set relations, $\in, \notin, \subseteq, =, \neq$, and three operators $(\cup, \cap, \setminus)$. It also provides the cardinality operator, which is important because it links the set constraints to the numerical constraints. Cardinality constraints are one way that information propagates between the CLPS and CLP(FD) solvers. The API of the solver includes procedures for adding each kind of primitive constraint, *query* procedures for finding information about the domains of variables, and *labelling* procedures for iterating through all possible values of a given variable.

Internally, the CLPS solver represents the domain of each variable using one or more of the following representations: *undefined, min-max (bounded)* and *enumerated*. Initially, the domain of a new domain variable (say X) is set to *undefined*. When a membership constraint (e.g., $X \in PID$) is added, an *enumerated* domain is used to record all the possible values that X can take. When a subset constraint (e.g., $X \subseteq Expr$ or $Expr \subseteq X$) , a *min-max* domain is used to record the lower bound and upper bound of X.

Usually in Constraint Satisfaction Problems (CSP), solvers allow only *valued domains*, where domains are sets of *values*. CLPS extends this by introducing the notion of *V-CSP* [BLP02], which allows *variables* to appear in the domains of other variables. This means that more complex rules are needed to manipulate these symbolic domains, but the advantage is that a higher level of abstraction can be achieved. CLPS allows variables within min-max domains and within enumerated domains under the condition that all elements are different in a *variable domain*. When two variables are unified, the intersection of their valued domains can be calculated immediately. However, for the variable domains representation, it is not always possible to combine the two symbolic domains, so the variable domain stores a *set* of domains and the true domain of the variable is the *intersection* of these domains.

The execution model of CLPS can be viewed as a transition system, where transitions represent reduction, inference and consistency rules [JM94]. The consistency algorithm used by CLPS ensures partial consistency, using on one hand the domain representations, and on the other hand the constraints store. To maintain domain consistency, CLPS combines arc-consistency techniques derived from AC3 [Mac77] and interval-narrowing over valued domains. So it is possible to have some elements

in domains not removed by these techniques, because only couple of variables are checked. The solver uses similar methods to reduce *variable domains* except that instead of manipulating one domain per variable, it manipulates a conjunction of domains (enumerated and min-max).

The constraint store is used for two purposes. Firstly, it allows the propagation of domain reductions, which enables consequences of constraints to be deduced. Secondly, it allows the solver to detect some semantic inconsistencies, which is crucial for pruning the search space. The following example shows how propagation works when constraints are acquired by the solver. In this example, B, C, D, G, X, Y are variables, Val_Dom_X represents values included in the valued domain and Var_Dom_X represents the conjunction of variable domains.

	Acquired constraints	Val_Dom_X	Var_Dom_X
1	$X \in \{B, C, D, Y\}$	*undef*	$\{\{B, C, D, Y\}\}$
2	$X \in \{B, C, D, 2\}$	*undef*	$\{\{B, C, D, Y\}, \{B, C, D, 2\}\}$
3	$X \in \{3, 4, G\}$	*undef*	$\{\{B, C, D, Y\}, \{B, C, D, 2\}, \{3, 4, G\}\}$
4	$X \in \{1, 2, 3, 4\}$	$\{1, 2, 3, 4\}$	$\{\{B, C, D, Y\}, \{B, C, D, 2\}, \{3, 4, G\}\}$
5	$Y \neq X$	$\{1, 3, 4\}$	$\{\{B, C, D\}, \{3, 4, G\}\}$
6	$G = 5$	$\{3, 4\}$	$\{\{B, C, D\}\}$

Constraints from line 1 to line 4 build up *variable domains* and the valued domain for X. On line 5, one of the *variable domains* is reduced from $\{B, C, D, Y\}$ to $\{B, C, D\}$. When a domain is included in another domain, the solver uses the following rule: $V \in Dom_1, V \in Dom_2, Dom_1 \subseteq Dom_2, T = Dom_2 \setminus Dom_1 \Rightarrow V \notin T$. In this case, due to the fact that all elements are different in a *variable domain*, it infers $X \notin \{2\}$, which is equivalent to $X \neq 2$. On line 6, one of the *variable domains* becomes valued, which causes the valued domain to be recalculated by intersection: $Dom\_Val_X = \{1, 2, 3, 4\} \cap \{3, 4, 5\}$.

After all 6 constraints have been added, we can conclude that $X = 3$ or $X = 4$, and that at least one of B, C or D must also equal 3 or 4. If a unique solution is not obtained after all constraints have been added, a user of CLPS-BZ typically uses the labelling procedures to explore the possible solutions, one at a time.

4.2 The Reducer

The Reducer reduces BZP predicates into the basic constraints of the solver CLPS. The most important procedure in the reducer API is `add(`P`)`, which conjoins the predicate P with the current constraint store, by reducing P into primitive constraints which are added into the constraint store. The reducer API also includes procedures for adding and deleting variables, so that the association between each specification variable and the corresponding Prolog constraint variable can be maintained.

The reduction of predicates to constraints is specified by a function ϵ, which takes a predicate as input, and returns a constraint as output. Generally, we have:

$$\epsilon(\text{Expression1 Operator Expression2}) = \\ \epsilon(\text{Expression1}) \text{ Constraint\_Operator } \epsilon(\text{Expression2})$$

In the Process Scheduler example, the precondition of the *New* operation is pp /: active \/ ready \/ waiting. This can be translated into constraints as follows (_Active, _Ready, _Waiting, _PP are Prolog variables that represent the constrained values of the specification variables active, ready, waiting and pp).

ϵ(pp /: active \/ ready \/ waiting)
= ϵ(pp) nin ϵ(active \/ ready \/ waiting))
= _PP nin ϵ(active) union ϵ(ready \/ waiting)
= _PP nin _Active union ϵ(ready) union ϵ(waiting)
= _PP nin _Active union _Ready union _Waiting

where "nin" and "union" are operators of the solver CLPS. They respectively represent non membership and the union of two constrained values.

4.3 The Executer

The Executer is used to animate a machine specified in a BZP file. The executer API provides commands for loading a particular BZP file, initializing the machine specified in that file, and executing the operations of that machine.

To initialize a machine, it simply declares all the state variables of the machine then sends all the initialization predicates to the reducer.

The execution of an operation takes place in two stages:

- validation of preconditions, then
- execution of postconditions.

First, all the precondition predicates are added. If no inconsistencies are found, all the postcondition predicates are added, and the primed state variables become the new machine state. If inconsistencies are found, the operation is not executed.

5 The Test Generation Modules

The generation of boundary Goal is performed in two main stages. The first is computed a set of *boundary goals* from the Disjunctive Normal Form (DNF) of the specification, The secound is instantiated each boundary goal into a reachable *boundary state*. Finally, the boundary states of the specification are used as a basis to generate test cases.

5.1 Generation of Boundary Goals

For test generation purposes, the postcondition of each operation is transformed into DNF, obtaining $\bigvee_j Post_j(op)$. The DNF transformation is not in general purpose, but only computes one by one operation and the conditions described in it. Then we project each of these disjuncts onto the input state, using the formula [LPU02b]:

$$(\exists\, inputs, state', outputs \bullet Pre \wedge Post_j)$$

We call these state subsets *precondition sub-domains*. The aim of boundary goal generation is to find boundaries within each of these precondition sub-domains.

In practice, the CLPS-BZ solver reduces each sub-domain to a set of constraints. For example, when the *New* operation of the Process Scheduler example is translated to BZP format, and each of its precondition and postcondition is transformed into DNF, we get the following predicates:

$$pre(new)_1 == pp \notin waiting \cup ready \cup active$$
$$post(new)_1 == prime(waiting) = waiting \cup \{pp\} \wedge$$
$$prime(ready) = ready \wedge prime(active) = active \wedge .. \wedge$$
$$prime(active) = \{\} \Rightarrow prime(ready) = \{\}$$

which is reduced by the BZ-TT solver to the predicate PS such that:

$$PS == \{Inv \wedge \#(waiting \cup ready \cup active) < \#PID\}$$

We compute boundary goals on the basis of the partition analysis by minimization and maximization using a suitable metric function chosen by the test engineer. (e.g., minimize or maximize the sum of the cardinalities of the sets). According to the optimization function, this results in one or several minimal and maximal boundary goals for each predicate.

Given the invariant properties Inv, a precondition subdomain predicate PS_i, a vector of variables V_i which comprises all the free state variables within PS_i, and f an optimization function, the boundary goals are computed as follows:

$$BG_i^{min} = minimize(f(V_i), Inv \wedge PS_i)$$
$$BG_i^{max} = maximize(f(V_i), Inv \wedge PS_i)$$

The optimization function $f(V_i)$, where V_i is a vector of variables $v_1 \ldots v_m$, is defined as $g_1(v_1) + g_2(v_2) + \ldots + g_m(v_m)$, where each function g_i is chosen according to the type of the variable v_i.

For example, from the predicate PS of the process scheduler example, boundary goals BG^{min} and BG^{max} are computed with the optimization function $f(V_i) = \sum_{v \in V_i} \#v^2$. The result of constraint solving is a set of constraints on the cardinalities of the set variables *waiting*, *ready* and *active* such that: $BG^{min} = \{waiting = \{\} \wedge ready = \{\} \wedge active = \{\}\}$, and $BG^{max} = \{waiting = \{X_1\} \wedge ready = \{X_2\} \wedge active = \{X_3\}\}$ where $(\forall\, i \cdot X_i \in \{p_1, p_2, p_3, p_4\})$ and $(\forall\, i \cdot i \neq j \Rightarrow X_i \neq X_j)$. It should be noted that other optimization functions could be used ($\sum_{v \in V_i} \#v$, $\sum_{v \in V_i} \sqrt{v}$,...).

5.2 Test Construction

This section describes the generation process of each test case, which is comprised of a preamble, a body, an identification and a postamble part [LP01].

5.2.1 Preamble Computation

Each boundary goal is instantiated to one or more reachable boundary states by exploring the reachable states of the system, starting from the initial state. The CLPS-BZ solver is used to simulate the execution of the system, recording the set of possible solutions after each operation. A best-first search [Pre01] is used to try to reach a boundary state that satisfies a given boundary goal. Preamble computation can thus be viewed as a traversal of the reachability graph, whose nodes represent the constrained states built during the simulation, and whose transitions represent an operation invocation. A consequence of this path computation is that state variables which are not already assigned a value by the boundary goal, are assigned a reachable value of their domain.

Some boundary goals may not be reachable via the available operations (this happens when the invariant is weaker than it could be). By construction, every boundary goal satisfies the invariant, which is a partial reachability check. In addition to this, we bind the search for the boundary state during the preamble computation, so that unreachable boundary goals (and perhaps some reachable goals) are reported to the test engineer as being unreachable. If all boundary goals in a precondition sub-domain PS are unreachable, we relax our boundary testing criterion and search for any preamble that reaches a state satisfying PS. Finally, the test engineer can

add and delete boundary goals, to customize the test generation, or to satisfy other test objectives.

5.2.2 Input Variable Boundary Analysis and Body Computation

The purpose of the body computation, or *critical test invocation*, is to test, for a given boundary state (a preamble), all the update operations, with all boundary values of their input variables. For the boundary values which satisfy the precondition, we get a positive test case, otherwise we get a negative test case. Note that, from the same preamble and boundary state, several bodies are usually obtained for each operation, with differing input values.

The process of boundary analysis for input variables is similar to that for state variables, except that invalid input values are kept, which is not the case for unreachable boundary states. Given an operation Op with a set of input variables I_i and a precondition Pre, let BG_i be a boundary goal. Note that BG_i is a set of constraints over the state variables, typically giving a value to each state variable. Then, given f an optimization function chosen by the test engineer, the input variable boundaries are computed as follows:

– for positive test cases:

$$minimize(f(I_i), Pre \wedge BG_i)$$
$$maximize(f(I_i), Pre \wedge BG_i)$$

– for negative test cases:

$$minimize(f(I_i), \neg Pre \wedge BG_i)$$
$$maximize(f(I_i), \neg Pre \wedge BG_i)$$

5.2.3 Identification and Postamble

The identification part of a test case is simply a sequence of all observation operations whose preconditions are true after the body. The postamble part is computed similarly to the preamble, using best-first search.

6 Related work

The BZ-TT proposal follows an important stream of work in test generation from formal specification, particularly from model-based specification. But despite the enormous popularity of boundary-value testing strategy for black-box testing in the software practitioner guides, this approach has not been widely investigated for automatic specification-based test generation. But boundary-value analysis is related to partition analysis which has been the subject of systematic research since the early testing literature [Mye79] [WO80] [OB88].

Dick and Faivre [DF93] present an approach for partition analysis by computing a Disjunctive Normal Form - DNF - both for input variables and system states. The idea of automatic partition analysis was already present in the work of Gaudel et. al. [BGM91, DGM93] by unfolding axioms from an algebraic specification. In TTF [Sto93, CS94], and extensions of it [PA01], partition analysis and DNF transformation are used as heuristics for manually defining test templates. In [LPU02a], we present a precise comparison between BZ-TT and TTF on the basis of the GSM 11-11 Standard case-study.

Various works [DF93, HP95, vABlM97, Hie97], use the partition analysis to build a Finite State Automaton - FSA - corresponding to an abstraction of the reachability graph denoted by the specification. Test cases are then generated by finding a

path (or several) "*which traverses every transition with the minimum number of repetitions*" [DF93]. Unfortunately, this method is not easily automated, because it is difficult to choose an appropriate abstraction of the state space to generate the FSA. In the BZ-TT approach, due to the transformation of the B or Z specification into a constraint system, the FSA is never explicitly computed.

During the last few years, the use of constraint logic programming for test generation has seen growing interest [Pre01]. For code-based test generation, Gotlieb et. al. [GBR00] present a framework where a system of constraints is built from a Static Single Assignment form of the source code (for C programs). The resolution of the constraint system allows finding a path in the control flow graph to sensitize a given point (statement or decision) in the source code. Meudec [Meu01] uses CLP for symbolic execution of Ada code in order to generate tests. In specification-based test generation, Marre et. al. [MA00] interpret LUSTRE specifications in terms of constraints over boolean and integer variables and solve them to generate test sequences. Pretschner et. al. [PL01] translate System Structure Diagrams and State Transition Diagrams into Prolog rules and constraints to allow symbolic execution of the specifications and thus test generation. Van Aertryck et. al. [vABlM97] use DNF and constraint solving to generate an FSA and test sequences from a B-like specification, but this is not fully automated. Mostly, these techniques use existing constraint solvers (Boolean and finite domains in general). In BZ-TT, due to the specificity of set oriented notations of Z and B, we developed an original solver able to treat constraints over sets, relations and mappings. This solver co-operates with the integer finite domain solver.

7 Conclusions and future work

We have presented an environment for automatic boundary-value testing from set-oriented formal specification notation. This environment is strongly based on a constraint technology which offers specialized support for the B and Z toolkits (sets, relations, functions and sequences).

The major advantage of using this constraint technology is that it dramatically reduces the size of the search spaces during test generation, which allows the method to scale to larger applications.

Firstly, the constraint technology enables boundary goals to be found more efficiently, due to the constraint representation, which is based on min/max interval domain representation (both for sets and numeric variables). This means that some max/min limits can be obtained directly from the constraint store, without search. Even when labelling is used, the search space is reduced because the constraint system prunes the search tree.

Secondly, the boundary goals that we obtain are also a set of constraints, rather than a specific value for each variable. Typically, some key variables will have extremum values, but others will just be constrained. This allows the boundary goals to be more abstract, which makes it easier to find a preamble that reaches the goal. In other words, the constraint approach avoids premature commitment to precise values for every state variable, which could result in choosing unreachable states.

Thirdly, the search for a preamble is reduced in complexity. At each point during the best-first search, we must still consider every operation, but with constraints it is not necessary to consider every input value to each operation, because the entire set of allowable inputs can be represented as a set of constraints.

These three reductions in search space give this boundary-value test generation method good scalability, and allow us to handle applications that would not be possible without the constraint technology.

Typically, the number of boundary goals (and thus preambles) generated is proportional to the number of operations, while the number of tests generated is proportional to the square of the number of operations (because we try to test each operation at every preamble). The time taken to produce each test is more difficult to predict, because it depends upon the complexity of the constraints and the average length of each preamble. The first factor is bounded by the worst case complexity of the CLPS-BZ solver, which is $n^2 * nd * d$ [BLP02], where n is the number of variables, nd is the highest number of sub-domains and d is the size of the largest domain. The second factor depends upon how easily each boundary goal can be reached, which is highly specification dependent.

In the realistic industry case studies that we have completed, the specifications were quite complex, but the test generation time was acceptable on a typical personal computer. For example, in a recent case study on the Java Card transaction mechanism [BJLP02], the 15 page B specification contained 20 operations and 15 state variables, where some state variables had sizable domains, such as the backup memory variable, which was a total function from addresses $(0 \,..\, 255)$ to bytes. In this case study, 60 boundary goals were computed, producing around 4500 test sequences, taking around 15 hours of computation time on a 1GHz Pentium.

We have also used this technology to generate tests for other smart card applications. In the GSM 11-11 standard case study [LP01, BLLP02], the five page specification contained 11 operations and 12 variables, and produced 38 boundary goals and around 1000 tests, with a computation time of 50 minutes. In the transport ticket validation algorithm [CGLP01], there was only one operation, but it was complex (5 pages) with 15 variables, and generated around 100 test cases in 20 minutes. The number of test are huge but one human test must be 5 to 10 automatic tests.

Until now, the BZ-TT environment has been usable only by the development team, because it lacked user interfaces and integration between components. However, we are currently developing Java interfaces for animation and test generation and consolidating the CLPS-BZ solver and the overall environment. The objective is to produce a beta release for Windows and Linux by the end of 2002, under a free license for academic use. See the BZ-TT website [BZT02] for updated release information.

In the future, we will be focusing on several areas:

- Supporting the full B and Z notation, such as parts of the toolkits that are currently missing (sequences, trees etc.) and layered machines in B.
- Extending the CLPS-BZ solver for continuous domains to be able to address problems with real or floating variables.
- Extending the solver and the BZP notation to support other specification notations, such as state charts and the UML object constraint language (OCL).

Acknowledgment

This research was supported in part by a grant of the French ANVAR - Agence Nationale de Valorisation des Actions de Recherche. The visits of Mark Utting are supported under the French/New Zealand scientific cooperation program.

References

[Abr96] J-R. Abrial. *The B-BOOK: Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0 521 49619 5.

[ALL96] F. Ambert, B. Legeard, and E. Legros. Programmation en logique avec contraintes sur ensembles et multi-ensembles héréditairement finis. *Technique et Science Informatiques*, 15(3):297–328, 1996.

[BGM91] G. Bernot, M-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, November 1991.

[BJLP02] F. Bouquet, J. Julliand, B. Legeard, and F. Peureux. Automatic reconstruction and generation of functional test patterns - application to the Java card transaction mechanism (confidential). Technical Report TR-01/02, LIFC - University of Franche-Comté and Schlumberger Montrouge Product Center, 2002.

[BLLP02] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11.11 standard case-study. *Submitted to the Journal of Software Practice and Experience*, 2002.

[BLP02] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B – A constraint solver for B. In *Proceedings of the conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, Grenoble, France, April 2002.

[BZT02] The BZ-TT web site. `http://lifc.univ-fcomte.fr/RECHERCHE/TFC/Page_BZ-TT.html`, April 2002.

[CGLP01] N. Caritey, L. Gaspari, B. Legeard, and F. Peureux. Specification-based testing – Application on algorithms of Metro and RER tickets (confidential). Technical Report TR-03/01, LIFC - University of Franche-Comté and Schlumberger Besanon, 2001.

[CS94] D.A. Carrington and P.A. Stocks. A tale of two paradigms: formal methods and software testing. In *Proceedings of the Z user workshop (8^{th} Z User Meeting)*, pages 51–68, Cambridge, June 1994.

[DF93] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. *FME'93: Industrial-Strength Formal Methods*, LNCS 670:268–284, April 1993.

[DGM93] P. Dauchy, M-C. Gaudel, and B. Marre. Using Algebraic Specifications in Software Testing : A case study on the software of an automatic subway. *Journal of Systems and Software*, 21(3):229–244, June 1993.

[FJJV96] J-C. Fernandez, C. Jard, T. Jéron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In *Computed Aided Verification (CAV'96)*, volume LNCS 1102, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[GBR00] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In Springer-Verlag, editor, *Proceedings of the First International Conference on Computational Logic (CL'00)*, pages 399–413, London, UK, July 2000.

[Hie97] R. Hierons. Testing from a Z specification. *The Journal of Software Testing, Verification and Reliability*, 7:19–33, 1997.

[HP95] H.M. Hörcher and J. Peleska. Using formal specifications to support software testing. *Software Quality Journal*, 4(4):309–327, 1995.

[ISO] ISO. Information Processing Systems, Open Systems Interconnection. *OSI Conformance Testing Methodology and Framework – ISO 9646*.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming : A survey. *Journal of Logic Programming*, 19/20:503–582, May/July 1994.

[LP01] B. Legeard and F. Peureux. Generation of functional test sequences from B formal specifications – Presentation and industrial case-study. In 16^{th} *IEEE International Conference on Automated Software Engineering (ASE'01)*, San Diego, USA, November 2001.

[LPU02a] B. Legeard, F. Peureux, and M. Utting. A comparison of the BTT and TTF test-generation methods. In *ZB2002: Formal Specification and Development in Z and B*, volume LCNS 2272, pages 309–329, Grenoble, France, January 2002. Springer-Verlag.

[LPU02b] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In Lars-Henrik Eriksson and Peter Lindsay, editors, *Formal Methods Europe, 2002*, LNCS. Springer-Verlag, July 2002. To be published.

[MA00] B. Marre and A. Arnould. Test Sequence generation from Lustre descriptions: GATEL. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*, pages 229–237, Grenoble, France, 2000.

[Mac77] A.K. Macworth. Consistency in network of relations. *Journal of Artificial Intelligence*, 8(1):99–118, 1977.

[Meu01] C. Meudec. ATGEN: Automatic test data generation using constraint logic programming and symbolic execution. *The Journal of Software Testing, Verification and Reliability*, 11(2):81–96, 2001.

[Mye79] G.J. Myers. *The Art of Software Testing*. Wiley-InterScience, 1979.

[OB88] T.J. Ostrand and M.J. Balcer. The Category-Partition Method for Specifying and Generation Functional Test. *Proceedings of the ACM Conference*, 31(6):676–686, June 1988.

[PA01] K. Periyasamy and V.S. Alagar. A rigorous method for test templates generation from object-oriented specifications. *The Journal of Software Testing, Verification and Reliability*, 11(1):3–37, 2001.

[PL01] A. Pretschner and H. Lötzbeyer. Model Based Testing with Constraint Logic programming : First Results and Challenges. In *Proceedings of the 2nd ICSE Workshop on Automated program Analysis, Testing and Verification (WAPATV'01)*, pages 1–9, may 2001.

[Pre01] A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming. In BRICS, editor, *Proceedings of the Workshop on Formal Approaches to Testing of Software (FATES'01)*, pages 47–60, Aalborg, Denmark, August 2001.

[RO92] D.J. Richardson and S.L. Aha T.O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 105–118, Melbourne, Australia, May 1992. ACM Press.

[Spi92] J.M. Spivey. *The Z notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992. ISBN 0 13 978529 9.

[Sto93] P. Stocks. *Applying Formal Methods for Software Testing*. Phd thesis, The University of Queensland, 1993.

[Tre96] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software-Concepts and Tools*, 17(3):103–120, 1996.

[vABlM97] L. van Aertryck, M. Benveniste, and D. le Metayer. CASTING: a formally based software test generation method. *In 1st IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 99–112, 1997.

[WO80] E.J. Weyuker and T.J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions in Software Engineering*, 6(3):236–246, May 1980.

Using a Virtual Reality Environment to Generate Test Specifications

Stefan Bisanz Aliki Tsiolakis

University of Bremen, Germany
{bisanz,tsio}@informatik.uni-bremen.de

Abstract

The creation of test specifications that can be used for automated testing requires considerable skill in the field of formal methods. This article proposes a method that enables the development of test specifications by interaction with a virtual reality representation of the system under test. From these interactions, a formal test specification is generated. Its goal is to reduce the need for formal methods expertise and therefore concentrates on the knowledge of the application to be tested. It addresses domain experts who are not familiar with formal methods.

In this article, the character of the virtual reality model as well as its creation are discussed. Further, the generation of test specifications is explained. Statecharts are used as formal specification language for the result of the generation step.

1 Introduction

Testing embedded real-time systems is a complex and time-consuming task and thus a high level of automation is advisory. Ideally, the test team receives a complete and consistent system specification in a formal specification language. By use of a test tool, the automatic generation of test cases, the automatic execution of the test and the automatic evaluation can be supported based on this specification. This idea relies on two main preconditions: On the one hand, the existence of a formal system specification developed by the domain experts and, on the other hand, on the availability of an appropriate test tool.

However, most system specifications are informal, natural language descriptions of the system's behaviour and the test experts have to develop the necessary test specifications manually based on these system specifications. Since the formal specifications require considerable skills in the field of formal methods, the domain experts can usually not generate the test specifications themselves. Nevertheless, approriate

test tools can be found. For example, *RT Tester*[1] supports automatic test generation, execution and evaluation based on formal test specifications in Timed CSP. These Timed CSP specifications are then translated into labelled transition systems (LTS). RT Tester has been applied in many application areas – from avionics to railway controllers.

Thus, the process of automatic test execution and evaluation is sufficiently supported by existing test tools. To fill the gap, we want to introduce a new approach of generating formal test specifications by interacting with a virtual reality representation of the system under test (SUT) called the *Virtual Periphery*. The Virtual Periphery represents the functional interface of the SUT embedded into its environment. For example, consider as a SUT the fasten-seatbelt signs in an airplane that can be switched on or off automatically depending on the status of specific sensors or manually by a dedicated switch in the cockpit. The functional interface consists of interface objects like signs, switches or sensors which are represented as interactive objects in the Virtual Periphery. The additional non-interactive objects constitute the geometry that is not of functional relevance to the SUT but serve to model the SUT's environment. To specify that the fasten-seatbelt signs are switched on by toggling the switch, one has to interact with the corresponding interface objects. Thus, the approach assists the domain experts intuitively during the specification process because the interaction's semantics can easily be learned. Interactions with VR components are mapped to specification fragments of a formal specification language that can be composed into the complete test specification.

Nevertheless, this basic Virtual Periphery is not sufficient to specify specific concepts, e. g., parallelism, or alternative interactions. For example, it is not possible to express by these interactions that all fasten-seatbelt signs have to be switched on in parallel because it is not possible to perform several simultaneous actions in the Virtual Periphery. To reduce this drawback, it is necessary to enhance the Virtual Periphery with visual representations of commands, graphical menues, gesture interactions, or voice commands. Furthermore, the Virtual Periphery only reflects the "present point in time" while it is not possible to investigate the history of past interactions or the potential future events. In order to overcome this, we have chosen statecharts as a formal but as well graphical specification technique. Statecharts provide an alternative view on the test specification fragments and facilitate their understanding. This choice has essential impact on our specification approach because, on the one hand, it affects the semantics of the VR interactions and, on the other hand, it defines the maximum expressive power of the resulting specifications.

In the next section, we give a detailed overview about the Virtual Periphery and some possible enhancements. The representation of the test specification fragments as statecharts is described in section 3. Section 4 refers to the integration of the generated statechart specifications with RT Tester. In section 5, we discuss ways to automatically generate the Virtual Periphery.

[1]RT Tester has been developed by Verified Systems International GmbH (http://www.verified.de) in cooperation with University of Bremen.

2 Virtual Periphery

The Virtual Periphery is an incomplete simulation model of the SUT, i. e., it is a model of the real world that only contains those objects that are relevant for the SUT's interface. Additional non-functional geometry enriches the Virtual Periphery and serves to model the SUT's environment such that it resembles the real world. The interface objects consume inputs (e. g., switches, buttons, sensors), yield output (e. g., signs, loudspeakers) or process input as well as generate output (e. g., touch screens, buttons with back light). Each interface object can be in a number of different states, and only specific state changes are possible. For example, a specific two-state toggle's states are ON and OFF, and a specific indicator can yield the states RED, YELLOW and GREEN. Each interface object is associated with specific attributes which include the interface object's type. Returning to our previous example, the fasten-seatbelt signs in an airplane denote a specific type of sign which has the attributes SEAT ROW and AISLE. Thus, specific objects of the same type can be identified. Furthermore, similar interface objects can be grouped, e. g., applying the condition `type = "Fasten-Seatbelt Sign" and seat row` $>$ `20 and aisle = left` identifies all interface objects of type *Fasten-Seatbelt Sign* in the left aisle of an aircraft at seat row 21 or higher.

As interface objects of the same type share geometry, possible states and attributes, interface object templates can be provided by interface object libraries. General purpose libraries contain general switches, indicators, etc. while domain specific interface objects (e. g., the fasten-seatbelt signs) are defined in domain specific libraries. The interface object templates and thus the libraries are modelled manually and are utilised within the Virtual Periphery creation process (see section 5).

The use of our Virtual Periphery exceeds simple simulation. By interacting with interface objects the user generates test specification fragments. This means that specific state changes in one input interface object are reflected in state changes of other output (or input/output) objects. More precisely, the state changes in the input interfaces are initiated by interactions and are used as stimuli for the SUT. The expected reaction is represented by the correct output.

Interaction with interface objects takes place by navigating a *3D cursor* that looks like a human hand. By touching an interface object (i. e., more exactly by colliding the 3D cursor with it), it gets selected for further interactions. Moving or rotating the 3D cursor generates a basic test specification element.[2] What movement or rotation is appropriate depends on the type of the selected interface object, but it should conform to the *direct manipulation metaphor*. This direct manipulation changes the interface object's state and its visual representation. For example, a toggle switch changes its state according to the rotation direction: the movement of the 3D cursor's fingertips chooses which part of the toggle switch is pressed down. Figure 1 shows a 3D cursor and three-state toggle switches in the cockpit of an airplane. For an introduction to 3D interaction see [BKLP01] and for a general discussion on direct manipulation see [Shn83] and [HHN86].

[2] If a conventional mouse is used for interactions, there are two dimensional mouse interactions that correspond to these three dimensional ones.

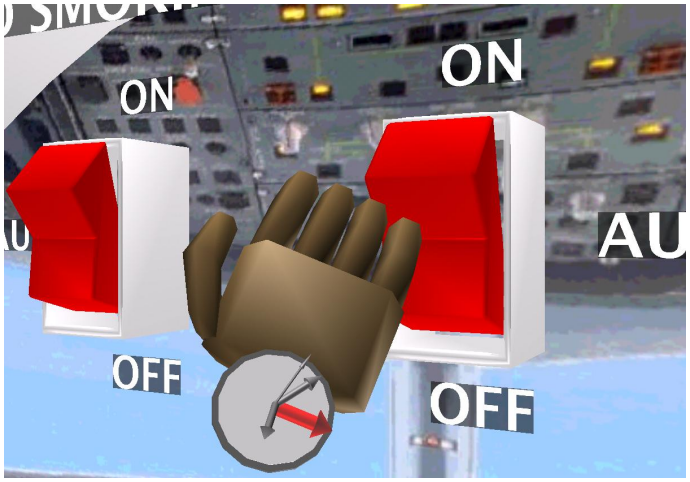


Figure 1: 3D cursor interaction with toggle switch

However, the direct manipulation metaphor is not appropriate for all interface objects. Consider interface objects that are not targets of tactile interaction in the real world (e. g., a sign or a sensor). Their state changes are selected using a graphical 3D menu within the Virtual Periphery that is triggered by a 3D metaphor of a mouse click, i. e., by a short movement of the 3D cursor's fingertips towards the interface object. An example of a 3D menu to select the state change of a sensor is given in Figure 2.

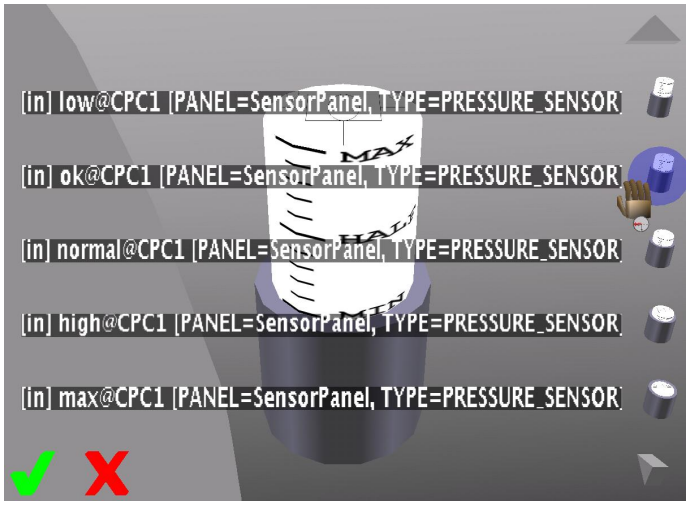


Figure 2: Selection of a sensor's state change using graphical 3D menu

Besides the central direct manipulation metaphor, we make use of a technique that integrates naturally within virtual reality: voice commands. While direct manipulation is used for interaction within the Virtual Periphery, speech input is used for *system control* and therefore for interaction with the Virtual Periphery itself.

In order to realize different semantics of direct manipulation interaction, the Virtual Periphery must provide different *interaction modes*. For example, a *causality* mode would enable specification fragments like

`if switch SW changes to state` ON`, then sign SI will be` LIGHTED,

and a *parallel* mode would enable

`all signs` SI_1 `...` SI_n `change to state` LIGHTED `in arbitrary order`

Note that speech based system control is superior to graphical system control because it does not interrupt the interaction within the Virtual Periphery. Simple *navigation* is also controlled by speech: predefined viewpoints can be activated such that the viewer[3] is immediately transported to the corresponding location within the virtual world.

Another feature supported by speech input is the *selection* of interface objects: in order to use several similar interface objects within a specification fragment, one selects them before interacting with one of them that then acts as a placeholder for all these objects.

While speech selection provides selection of currently visible interface objects, alternatives are mouse based selection on the one hand and attribute based selection on the other hand.

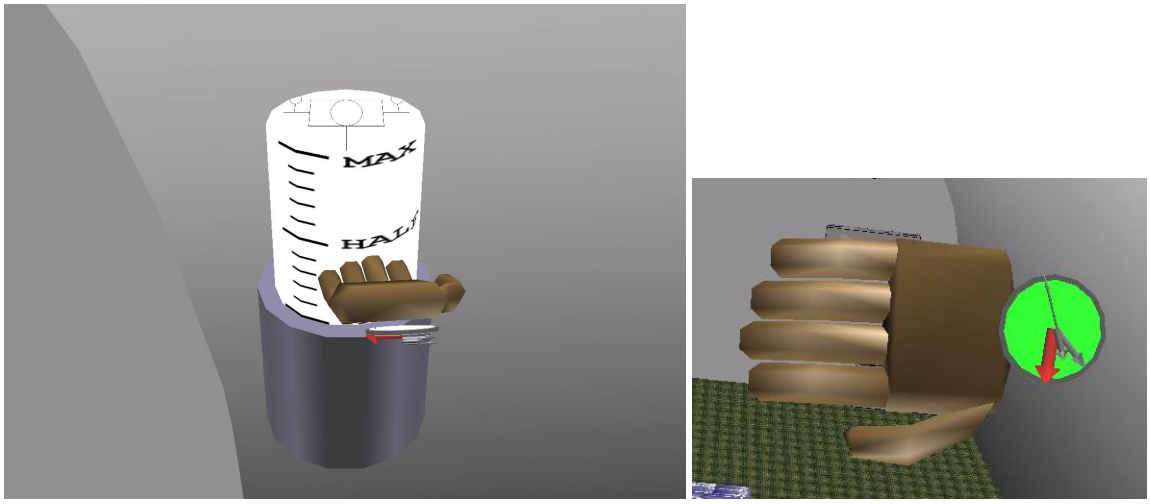


Figure 3: Reference gesture (left) and time gesture (right)

In order to switch specific interaction modes, *gestures* can be used. For example, the so-called *reference gesture* is a rotation of the 3D cursor so that it looks like an open hand, palm up. In combination with a subsequent direct manipulation interaction as described above, it references the complete interface object and defines the current specification context. The so-called *time gesture* will switch the specification mode so that further interactions are time dependant. Thereby, the 3D cursor which is equipped with a watch is rotated as if the user is taking a look on it. Both gestures are shown in figure 3.

[3]That is the person interacting with the Virtual Periphery.

3 Incremental Development of Statecharts by Interaction

Behaviour Template With each interface object a pre-defined *behaviour template* is associated: a statechart consisting of its possible states and appropriate transitions. The behaviour template of an interface object is created manually based on its interface and stored as an additional attribute of its template in the interface object library (see section 2). Note that the states in the behaviour template correspond to the informally described states of the interface object. Additionally, the behaviour template contains transitions between the states based on the interface description of the interface object. Considering a simple two-state switch as an example, the corresponding statechart contains two states ON and OFF. The state changes would be switching from ON to OFF and vice versa, therefore the statechart contains two transitions triggered by the events SWITCH.OFF and SWITCH.ON, respectively. Figure 4a shows the behaviour template of the two-state switch.

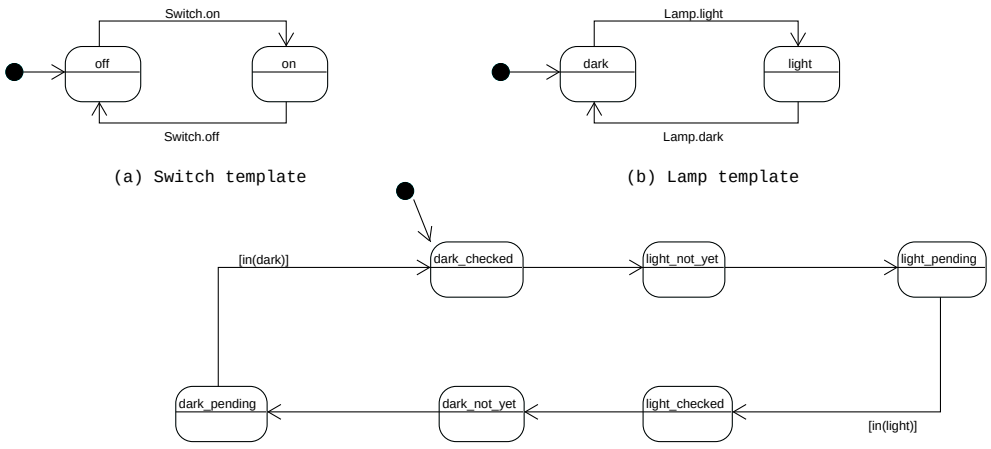


Figure 4: Behaviour template of switch and lamp

Additionally, the behaviour template can contain a checker component that is typically needed for our test purposes. As an example, consider a simple lamp or sign that provides outputs to the SUT's environment: while on the one hand the real lamp is in one of its states DARK or LIGHT (see figure 4b), we want to check if its state changes occur appropriately depending on certain events or conditions within the SUT. While these events and conditions are subject to the specification process, the checker component generally distinguishes valid and pending situations. Figure 4 shows the behaviour template of a lamp consisting of a statechart for the real lamp and the checker statechart. The latter is denoted in figure 4c and contains different states: The states DARK_CHECKED and LIGHT_CHECKED represent the correct (and checked) states and thus correspond conceptually to the states DARK and LIGHT in figure 4b. The states DARK_PENDING and LIGHT_PENDING represent the

situation when the event (or condition) to change the state has already occured but the correct reaction of the SUT has not been checked yet. Finally, DARK_NOT_YET and LIGHT_NOT_YET denote that after the occurence of the change event the SUT is not expected to react yet.

Interaction Scenario Let us now consider the interaction scenario for the following requirement:

```
The lamp must be dark, if the switch is off
and it must be lighted, if the switch is on.
```

The interface objects are a two-state switch and a lamp whereby the former is initially in state OFF and the latter is initially DARK. On template base, the application specific dependencies between switch, lamp and checker component are not yet defined. The following steps describe in a step-by-step manner how these dependencies are introduced to the templates of figure 4 for the concrete application context. Since only correct behaviour should be specified, the system state has to be consistent with respect to the current specification context.

1. The process is started by choosing the specification context, i. e., this is the lamp in the example. Therefore, we navigate to the lamp within the Virtual Periphery by issuing the voice command VIEW LAMP. By applying the reference gesture and touching the lamp, we reference the lamp's behaviour template (i. e., the corresponding statechart).

2. To define the trigger for the state change of the context object, we navigate to the trigger object, i. e., in this example to the switch, by voice command VIEW SWITCH. The following manipulation of the switch – a rotation of the 3D cursor – changes the state of the switch to ON. This state change is represented by a transition in the switch's statechart.

3. To define the effect of the trigger on the context object, the current state of the context object is considered as the source state for a transition. Since the lamp cannot be manipulated directly, the target state (i. e., the state LIGHT) has to be selected using a 3D menu. Although the corresponding 3D menu provides the states LIGHT and DARK (i. e., states defined in the lamp's statechart), the checker statechart is affected. The transition with source state DARK_CHECKED and target state LIGHT_NOT_YET is labelled with the guard IN(SWITCH.ON).

4. The same steps have to be applied for specifying the second part of our specification, respectively. Thereby, the transition from state LIGHT to target state DARK_NOT_YET is labelled with IN(SWITCH.OFF).

In the above specification scenario, the resulting specification fragment is only causally coherent but does not contain any timing constraints. Moreover, the checker might stay in state LIGHT_PENDING without detecting any errors. Since the system specifications usually imply specific requirements to react in a certain time interval,

it is necessary to apply as well the time gesture (see section 2). Thus, before starting the above described specification process, the time gesture is applied to indicate that the following specification mode is time dependant. The effect of the timed specification mode is that when the trigger of a state change is defined, additional transitions and labels are inserted in the checker statechart that check the lower and upper bounds of the time interval.

In the above example, in order to check the upper bound two transitions are added – one at state LIGHT_PENDING and the other one at state DARK_PENDING. The transitions are labelled with a timeout event TM(EN(LIGHT_PENDING),T_2) and TM(EN(DARK_PENDING),T_4), respectively, and a resulting ERROR action.[4] The events are triggered T_n time units after the last entry to state LIGHT_PENDING or DARK_PENDING, respectively. The lower bound check of the state change to LIGHT is realised by the timeout event TM(EN(LIGHT_NOT_YET),T_1) in combination with a new transition between LIGHT_NOT_YET and LIGHT_CHECKED. The latter can only be taken before T_1 time units elapse and therefore results in an ERROR action. The lower bound check of the state change to DARK is specified in a similar way. The concrete timer values T_n cannot be set directly, therefore default values are used that have to be further adjusted (e. g., by the use of a 3D menu in the Virtual Periphery). Note that the lower bound check as well as the upper bound check of the time interval can be omitted.

The resulting statecharts are shown in figure 5. In addition to the above mentioned statecharts for interface objects, another statechart is generated during the specification process. While interacting with the Virtual Periphery, the user triggers certain input interface objects (e. g., a switch) to change state. These user statecharts cannot be defined as behaviour templates in a library, since their states and transitions depend entirely on the test specification for the SUT.

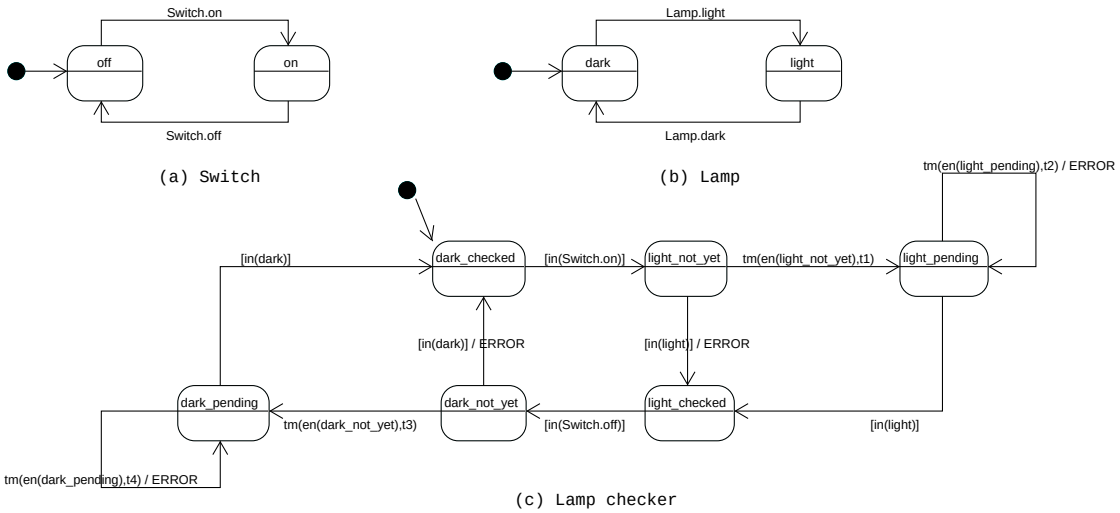


Figure 5: Instantiated and extended statecharts

[4] A more detailed error handling is necessary but is not discussed in this paper.

4 Testing with Statecharts

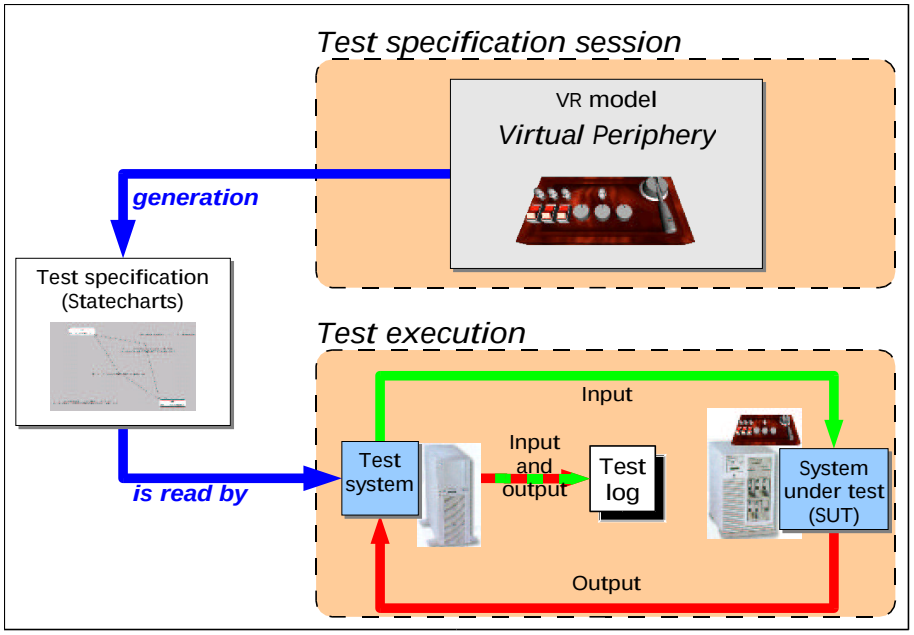


Figure 6: Test process overview

An overview about the complete test process is given in figure 6. While, on the one hand, the statecharts (as discussed in Section 3) are the result of a test specification session, they are, on the other hand, the input to the test system. We focus in our work on RT Tester. RT Tester generates and executes tests automatically by sending inputs to the SUT. The tests are evaluated on the fly based on the given inputs and the outputs coming from the SUT. See [Pel02] and [PT02] for application examples and [Pel98] for the theoretical background.

Since the RT Tester tool accepts test specifications as *labelled transition systems (LTS)* to allow the use of arbitrary formal test specification languages that can be translated into an LTS, we have to provide such a translation relation.

This translation depends on the statechart semantics used during the specification process. Different semantics are available for statecharts: Harel introduced statecharts in 1987 (see [Har87]) and gave a formal semantics in [HPSS87]. A variant described in [HN96] has been implemented in the STATEMATE tool[5]. Statecharts have as well been integrated in UML (see [Obj]) with a slightly different semantics. Other semantics have been discussed as well, and a comparision of different statechart semantics is given in [vdB94]. As well, different approaches for the translation of statecharts into LTS have been discussed, see e. g. [US94], [Lev96] and [Joh99]. Most variants are tailored to meet specific needs. We need a semantics that can at least deal with our timing constraints and which has a step semantics with a greedy approach. Nevertheless, we are currently investigating the specific needs of our approach with respect to the statechart semantics. Thus, we can yet neither provide

[5]STATEMATE is a commercial tool by i-logix (http://www.ilogix.com) and is actually applied to industrial projects.

a complete algorithm for the translation nor a precise semantics for the statecharts used in our approach.

5 Virtual Periphery Generation and Reuse

One crucial point within our approach is the creation of the Virtual Periphery itself. Since the manual generation of the Virtual Periphery is a very time consuming and thus expensive task and moreover highly SUT dependant, it is desirable to minimise any manual effort to create the Virtual Periphery. Hence, we want to discuss in the following an approach to support the generation of the Virtual Periphery. Figure 7 gives a first overview.

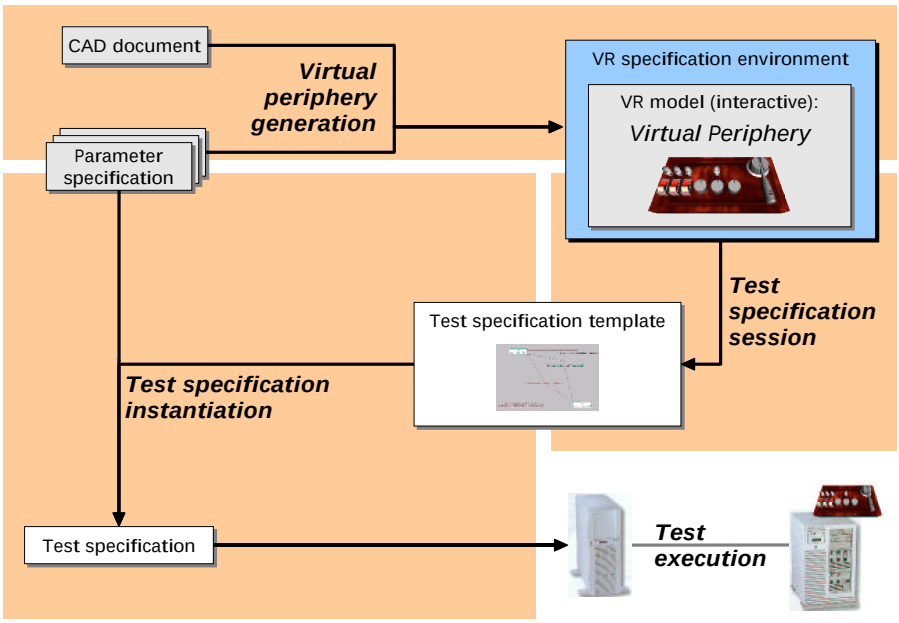


Figure 7: Virtual Periphery generation and test process

In most application areas, there are static geometry models of the SUT's environment and/or the SUT itself. This collection of documents – typically CAD documents – can be converted straight-forward into an appropriate virtual reality model.

Additionally, some SUTs are equipped with some kind of *parameterisation module* in order to configure system features. Consider the *Cabin Intercommunication Data System* that is used within Airbus airplanes. It contains the so-called *Cabin Assignment Module* that parameterises for example, how many attendant handsets are available in the cabin and to which controllers they are connected. A similar example is the number and mapping of passenger service units[6] to seat rows and aisles. Although the parameterisation modules are typically domain specific, once an evaluation is realised it can be used to generate appropriate variants of the Virtual Periphery depending on the specific parameter values.

[6] A passenger service unit is a collection of signs, keys, lamps, ... above the passenger's seat.

Typically, only an intermediate format – a non-interactive virtual reality model – can be derived automatically from the given documents. Hence, the interface objects have to be inserted or replaced manually by interactive objects in order to specify the tests as discussed in the previous sections. This process can be supported by libraries of interface objects which contain for each type of interface object its geometry as well as the corresponding behaviour template (see sections 2, 3).

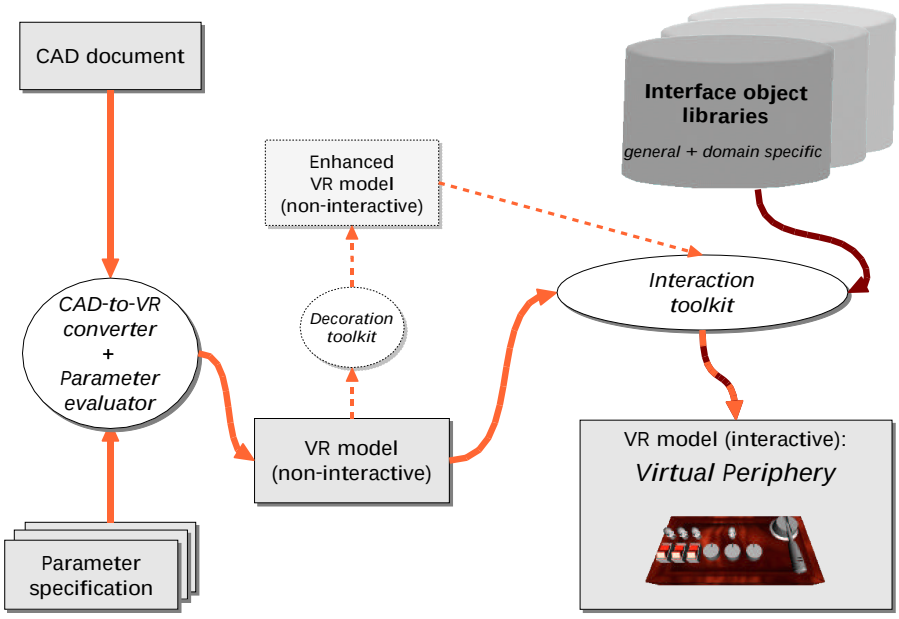


Figure 8: Detailed Virtual Periphery generation

Figure 8 gives a detailed view of the Virtual Periphery generation. It also contains an optional step via an enhanced non-interactive model. This may be desired if the Virtual Periphery is supposed to contain decorations like textures, which usually are not provided within the CAD documents. Enhancing the model is a manual activity similar to the insertion of interface objects.

While the previously described generation approach itself reuses the CAD documents and the parameterisation module (i. e., documents generated not specifically for the purpose of testing), it is as well possible to reuse parts of the test specification fragments based on concrete parameter values. Considering, for example, the reading lights in an airplane, all reading lights can be switched on or off by a central button (using the selection mechanism described in Section 2 to select all reading lights). Additionally, each reading light can be switched on or off by a toggle switch above the passenger seat. Nevertheless, it is neither desirable to specifiy this specification part for each reading light separately nor should it be necessary to specify the behaviour of all reading lights once again, if the parameter value denoting the number of reading lights has changed. In contrast, a test specification template could be used which is instantiated with a concrete set of parameter values before testing (i. e., more precisley before generating the LTS). This approach is visualised in the lower left part of Figure 7 focussing on the test specification instantiation based on the generic test specification template and the concrete parameter values defined in the parameterisation module.

6 Conclusion

This article proposed an approach to facilitate the creation of formal test specifications providing interactive virtual reality components. The approach addresses domain experts who are not familiar with formal specification languages and allows them to create basic test specifications quickly and intuitively. Nevertheless, the person interacting with the Virtual Periphery should have a precise understanding of concepts like parallelism and the sequencing of events.

Thus we expect the benefit of our approach to be:

Simple specifications Simple specifications can be developed without applying elaborate concepts and thus without a detailed understanding of the underlying concepts of the formal specification language.

Team development Within a test team, domain experts and test experts can cooperate to extend the simple specifications.

Introduction to formal specification languages Additionally, our approach can be used to become familiar with formal specification languages in an intuitive way and thus to gain necessary expertise in it. Eventually, the expert will then even prefer to define the test specifications using directly the formal specification language.

The Virtual Periphery described in this article is partially implemented using Java3D which is an API for the general purpose, object oriented programming language Java. It is proved to be superior to VRML which we used during earlier efforts (see [BF99], [PBFE99]), because Java3D allows more flexible modelling. For further information concerning Java, Java3D and VRML see [GJS97], [Jav00] and [VRM97].

One main design guideline during the implementation is the use of conventional personal computers without extraordinary input or output devices. Hence, all interaction must be possible by mouse, keyboard and low cost microphone. Our 3D cursor is carefully designed to be used with the mouse to allow movements and rotations to be gained from two-dimensional mouse movements combined with so-called modifier keys (e. g., pressing of mouse buttons). Furthermore, the output has to be appropriate for conventional monitor screens and stereo pairs of speakers. Note that this is the reason why no haptical output like force feedback is available with the Virtual Periphery.

Nevertheless, it is possible to enhance the virtual reality feeling with special equipment. There is no inherent restriction of our 3D cursor to be used with the mouse, so that alternatively a data glove could be used. For visual output, a head mounted display as well as stereoscopic viewing solutions can be applied. As virtual reality audio enhancement, dolby surround or similar techniques are available. To gain an overview about virtual reality equipment refer for example to [MG96].

Future work will include the definition of a formal statecharts semantics and a corresponding translation to LTS that can be used by the RT Tester tool. Since the

formal language is exchangeable as far as there is an appropriate LTS representation, further evaluation of appropriate formal specification languages is planned. In particular, we will consider hybrid automata (see [Hen96]) that enable modelling of continuous behaviour and are in this respect more expressive than statecharts. However, the chosen language has essential impact on the interaction's semantics within the virtual reality.

Another point that is subject to further research concerns the test evaluation that is so far based on the generated test specification. A way to map an event from the test log of a test run to the corresponding Virtual Periphery representation would be valueable in order to interpret errors and warnings or even to find inconsistencies within the test specification itself. Since the test execution is based on labelled transition systems, this is not a trivial task. Even the mapping to the corresponding part of the statechart is non-trivial.

Finally note that although we focus on developing test specifications, our approach is not restricted to this kind of specifications but can be expanded to more general specifications.

References

[BF99] Stefan Bisanz and Ingo Fiß. Grafischer Entwurf von CSP-Spezifikationen für den Test eingebetteter Echtzeitsysteme. Master's thesis, Universität Bremen, February 1999.

[BKLP01] D. Bowman, E. Kruijff, J. LaViola, and I. Poupyrev. An Introduction to 3D User Interface Design. *Presence: Teleoperators and Virtual Environments*, 10(1):96–108, 2001.

[GJS97] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997. http://java.sun.com/docs/books/jls/html/.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[HHN86] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User Centered System Design: New Perspectives on Human-Computer Interaction*, pages 87–124. Erlbaum, Hillsdale, NJ, 1986.

[HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings, Symposium on Logic in Computer Science*, pages 54–64. The Computer Society of the IEEE, June 1987. Extended abstract.

[Jav00] The Java 3D^{TM} API Specification. Version 1.2. http://java.sun.com/products/java-media/3D/index.html, April 2000.

[Joh99] Sebastian John. Zur kompositionalen Semantik von objektorientierten Statecharts. Master's thesis, Technische Universität Berlin, August 1999.

[Lev96] Francesca Levi. A process language for statecharts. In *Logical and Operational Methods in the Analysis of Programs and Systems*, pages 388–403, 1996.

[MG96] Tomasz Mazuryk and Michael Gervautz. Virtual reality - history, applications, technology and future. Technical Report TR-186-2-96-06, Vienna University of Technology, Institute of Computer Graphics and Algorithms, A-1040 Wien, February 1996. Available as http://www.cg.tuwien.ac.at/research/TR/96/TR-186-2-96-06Abstract.html.

[Obj] Object Management Group (OMG). *OMG Unified Modeling Language, Specification*. Available at http://www.omg.org/uml/.

[PBFE99] J. Peleska, S. Bisanz, I. Fiß, and M. Endreß. Non-Standard Graphical Simulation Techniques for Test Specification Development. In H. Szczerbicka, editor, *Modelling and simulation: A tool for the next millenium. 13th European Simulation Multiconference 1999*, volume 1, pages 575–580, Delft, 1999. Society for Computer Simulation International.

[Pel98] Jan Peleska. Testing reactive real-time systems. Tutorial, held at the FTRTFT '98, Denmark Technical University, Lyngby, 1998. Updated revision. Available as http://www.informatik.uni-bremen.de/agbs/jp/papers/ftrtft98.ps.

[Pel02] Jan Peleska. Formal methods for test automation - hard real-time testing of controllers for the airbus aircraft family. In *Proc. of the Sixth Biennial World Conference on Integrated Design & Process Technology (IDPT2002), Pasadena, California*. Society for Design and Process Science, June 2002. To appear.

[PT02] J. Pelska and A. Tsiolakis. Automated Integration Testing for Avionics Systems. In *Proceedings of the 3rd ICSTEST – International Conference on Software Testing*, April 2002.

[Shn83] Ben Shneiderman. Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8):57–69, 1983.

[US94] Andrew C. Uselton and Scott A. Smolka. A compositional semantics for statecharts using labeled transition systems. In *International Conference on Concurrency Theory*, pages 2–17, 1994.

[vdB94] M. von der Beeck. A comparison of StateCharts variants. In *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, pages 128–148, Berlin, 1994. Springer-Verlag.

[VRM97] VRML Consortium. The Virtual Reality Modeling Language. http://www.web3d.org, 1997. International Standard ISO/IEC 14772-1:1997, Part 1 – 3.

Towards a Formalization of Viewpoints Testing

Marius Bujorianu[1], Savi Maharaj[2], Manuela Bujorianu[2]

[1]Computing Laboratory, University of Kent,
Canterbury, Kent, CT2 7NF mcb8@ukc.ac.uk.
[2]Department of Computing Science and Mathematics,
University of Stirling, Stirling, Scotland, FK9 4LA, UK
{mlb, sma}@cs.stir.ac.uk

25 June 2002

Abstract. Test case generation from formal specifications is now a very mature field. Partial specification or viewpoints represents a co-operative approach in software specification. Although partial specification has a long history, only a little was done towards application of the methodology to formal test case generation. In this work we propose a categorical foundation of viewpoints oriented testing, obtaining a sound methods integration and a formal testing methodology for composite (heterogeneous) software systems. In particular, we plan to address the combination of specification based testing and test case generation from proofs.

Keywords: formal testing, viewpoints specification, category theory.

1. INTRODUCTION

The most challenging issue of software engineering still is the question how to master the complexity of the development of large software systems. For projects involving the specification and development of large systems, the structuring of their descriptions is crucial to the project's success. Traditionally, systems were decomposed according to functionality; modern approaches favour, in addition to this, decompositions according to "aspects" or "viewpoints". Also, these viewpoints may be views of the system's functionality from different participants. Viewpoints mean different perspectives on the same system, aspect oriented specifications written in different languages by teams having different backgrounds, etc. Of course, for an implementation we need an integration of all these formal descriptions, more specifically a minimal one, called unification.

The increasing importance of the viewpoint model in software engineering is exemplified by its use in the Open Distributed Processing (ODP) standard, OO design methodology, requirements engineering, software architecture, reverse engineering and the Unified Modeling Language (UML). ODP, for example, defines a viewpoint framework for specification of distributed systems using a fixed collection of five viewpoints: enterprise, information, computational, engineering and technology. We are particularly interested in the construction of a general model applicable to most of these application fields.

Different models for viewpoints have been proposed, like the VOSE framework [11] and the 'Development' approach [5]. Our approach follows the general model presented in [6] and [16]. In this work we extend the categorical framework developed for the homogenous case in [7]. Viewpoints are written using the specification language associated with a general logic [22]. The correspondences between viewpoints are expressed using diagrams, specifically spans from the correspondence specification to the viewpoints which are unified. Unification is given then by the pushout of the diagram.

Towards a Formalizationof Viewpoints Testing2

Although viewpoints oriented specification like techniques are used in a great variety of development technologies, there is no mature use of them in testing. Notable exceptions are MacColl and Carrington's framework MATIS [17] and Pemberton's VOCAL [23]. VOCAL: 'Viewpoint-Oriented verifiCation And vaLidation' is an application of the VOSE framework to software testing. Viewpoints have been applied for the identification and structuring of test deployment. Using such an organization helps to ensure that all important test perspectives are taken into account. In the VOCAL approach, viewpoints are classified in group viewpoints, verification viewpoints, validation viewpoints and quality viewpoints. The test technique applied within viewpoints provides test coverage. In contrast, MATIS is specification oriented and is much closer to the idea of our work.

Tests are derived from viewpoint specifications specifically, according to the formal language used by each team. Viewpoints could have different concepts for tests, different languages to describe them precisely, and different techniques to derive them. When we have a big heterogeneity of viewpoints, and implicitly of tests derived from them, we need a common definition and understanding of them, as well a set of tests for the whole system description (i.e. unification).

We suppose the reader is familiar with the basics of category theory. The space limit of this work doesn't allow a background presentation of the category theory, which has been used. We recommend the book [1] for the necessary background information.

The paper is structured as follows. In the next section we give a brief summary of a categorical and logical approach to software specification. In this section we study also the unification process and the development relations involved. In Section 3 we sketch a formal theory for composite viewpoints. In Section 4 we present the formal testing theory for axiomatic specifications as application introduced and developed by M.C. Gaudel [13]. In Section 5 the formal frameworks are applied to unify the tests generated from the viewpoints in order to obtain tests for the whole system. The partial conclusions of this formal experiment are sketched in the last section.

2. Axiomatic Viewpoint Specification

This section defines the formal concept of viewpoints used in the rest of the paper. We stress here especially the logical and categorical aspects of formal software specifications.

2.1. Categorical Specification Logics. The categorical logic we describe in this paragraph is based on the papers [**??**] and [22].

- **Syntax**.

Vocabulary : is given by a category of signatures **SIGN**. Associated with the category **SIGN** we have a functor $Sort : \mathbf{SIGN} \to \mathbf{SET}$, called the sort functor for **SIGN**; for any signature Σ , the elements $Sort[\Sigma]$ (or S) are called the sorts of Σ. By $\mathbf{SIGN}^{Var}$ we denote the category where an object is a pair (Σ, X) where X is a $Sort(\Sigma)$-sorted set whose elements are called *variables*, and a morphism from (Σ, X) to (Σ', X') is a pair (σ, v) with $\sigma : \Sigma_1 \to \Sigma_2$ a morphism of signatures and $v : X_1 \to (X_2)_{|\sigma}$.We call an object in $\mathbf{SIGN}^{Var}$ a *signature with variables*. We often ask for this category to be cocomplete. For the case studies we will consider in our work, **SIGN** is cocomplete.

Formulas are given by a functor $Syn : \mathbf{SIGN} \to \mathbf{SET}$, assigning to each signature Σ in **SIGN** a set of $\Sigma-$sentences $Syn(\Sigma)$.
We do not use here any specific language. We will introduce the syntax gradually, as we will use it.

- **Model Theory**

The interpretation is given by

i) a functor called interpretation functor $Int : \mathbf{SIGN} \to \mathbf{CAT}^{op}$ contravariant in **SIGN**, which gives to each signature Σ in **SIGN** a small category $Int(\Sigma)$ of $\Sigma-$models. These $\Sigma-$models provide the interpretations for the names in the vocabulary. Thus we can suppose that there are some concrete entities, to which sentences can refer. For any signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ in **SIGN**, there is a functor $Int(\sigma) : Int(\Sigma_2) \to Int(\Sigma_1)$ is called the reduct functor and denoted by $-_{|\sigma}$.
ii) a family of **Logical Satisfaction Relations** $\models = \{\models_\Sigma, \Sigma \in \mathbf{SIGN}\}$, $\models_\Sigma \subseteq |Int(\Sigma)| \times Syn(\Sigma)$, such that we have the **Logical Satisfaction Condition** $M_2 \models_{\Sigma 2} Syn(\sigma(a_1)) \Leftrightarrow Int(\sigma(M_2)) \models_{\Sigma 1} a_1$is fulfilled for all signatures morphisms $\sigma : \Sigma_1 \to \Sigma_2$ in **SIGN**, all $\Sigma_{\mathbf{2}}$-structures $M_2 \in |Int(\Sigma_2)|$ and all $\Sigma_{\mathbf{1}}$-axioms $a_1 \in Syn(\Sigma_{\mathbf{1}})$

- **Proof Theory**

Proof theoretic methods have been successfully applied in formal software development. Notable examples are the work in [3] for formal integration of VDM and B specification languages and in [19], where correctness proofs have been used as a source for tests generation. Anyway, in this work we will present only background introduction in proof theory for specification logics, necessary for a further implementations into a theorem prover.

A *consequence relation* (shortly CR) is a pair $(S, \vdash)$ where S is a set of formulas and $\vdash \subseteq \wp_f(S) \times S$ is a binary relation such that : (Reflexivity) $a \vdash a$; (Transitivity) If $\Gamma \vdash a$ and $a, \Gamma' \vdash b$ then $\Gamma, \Gamma' \vdash b$; (Weakening) If $\Gamma \vdash a$ then $\Gamma, b \vdash a$.

The closure operation on sets of formulas $\Lambda \subseteq S$ of a CR $(S, \vdash)$ is defined by

$$Cl_\vdash(\Lambda) = \overline{\Lambda} = \{a | \Gamma \vdash a, \Gamma \subseteq \Lambda\}$$

A set of formulas Λ is *closed* under $\vdash$ iff $Cl_\vdash(\Lambda) = \Lambda$.

Definition 1. *A theory (wrt $\vdash$) is a set of formulas closed under $\vdash$.*

Definition 2. *(CR morphism) A morphism of consequence relations $\tau : (S, \vdash) \to (S', \vdash')$ is a function $\tau : S \to S'$ (the translation of formulas) such that if $\Gamma \vdash a$ then $\tau(\Gamma) \vdash' \tau(a)$.*

Definition 3. ***(Category CR)*** *The consequence relations as objects and morphisms of consequence relations as arrows form a category, denoted* **CR**, *with identities and composition inherited from the category of sets.*

If $\tau : (S, \vdash) \to (S', \vdash')$ is a CR morphism and $\Lambda \subseteq S$ then $\tau(\overline{\Lambda}) \subseteq \overline{\tau(\Lambda)}$. Thus the image of a theory under a CR morphism is not always a theory. Moreover $\overline{\tau(\overline{\Lambda})} = \overline{\tau(\Lambda)}$.

Definition 4. ***(Logical system)*** *A logical system is a functor $L : \mathbf{SIG}_L \to \mathbf{CR}_L$, such that the deduction relation $\vdash_\Sigma$ is sound for the logical satisfaction relation $\models_\Sigma$ for every signature $\Sigma \in \mathbf{SIGN}$.*

A $\Sigma-$theory is a theory in $L(\Sigma)$. A $L-$theory is a $\Sigma-$theory for $\Sigma \in |\mathbf{SIG}_L|$. The category of theories will be denoted **TH**.

$\mathcal{D}_{\mathcal{CL}} = (\mathbf{SIGN}, Syn, \vdash)$ forms general deduction system and the structure $\mathcal{M}_{\mathcal{CL}}$=(**SIGN**, $Syn, Sem, \models$) gives the model theory (the *institution*) of the general logic $\mathcal{CL}$.

- **Examples of Specification Logics**

Example 5. *The logic $\mathcal{FOL}^{=}$: many-sorted first order logic with equality*

Signatures are many-sorted first order signatures. Models are many-sorted first order structures. Formulas are many-sorted first order formulas. Formula translations mean replacements of the translated symbols. Satisfaction is the usual satisfaction of a first order formula in a first order structure.

Example 6. *The logic $\mathcal{LTSHL}$: the labelled transition systems hidden logic*

Signatures are many-sorted first order signatures enriched with hidden sorts. Each hidden sort models the states of an lts. As used in hidden logic, we ask operations to be monadic on hidden sorts. Formulas are many-sorted first order sentences, models are simply many-sorted first order structures, satisfaction is defined also like in first order logic. Labelled transition systems (lts's) $(S, L, \rightarrow)$ are modelled as first order structure, where the support of a hidden sort models the state set S. Labels of L are behavioral formulas of the form *precondition* $\Rightarrow$ *action*. Actions are simply assignment statements as in imperative languages. The transition relation is a predicate $\rightarrow \subseteq S \times L \times S$. We use the notation $s - \{l\} \rightarrow s' = op(s)$ for the transition $s \stackrel{op,l}{\rightarrow} s'$, where *op* is an operation from a hidden sort to a hidden sort from the algebraic signature.

2.2. Viewpoint Specification and Unification in General Logics. A $\Sigma-$*presentation* in a general logic is a signature extended with a set of axioms. In most specifications an axiom consists of a set of variables and two terms of the same sort belonging to the term language of the signature with respect to the set of variables.

The semantics of a viewpoint is an element in a category, often an algebra. An algebra is *presented by a* presentation if it is denoted by the signature of the presentation and *satisfies the axioms* of the presentation.

Any specification formalism based on a general logic $\mathcal{L}$ determines a class of specifications, and then, for any viewpoint PSP, its signature $Sig[PSP] \in |\mathbf{SIGN}|$ and the collection of its models $Sem[PSP] \subset Int[Sig[PSP]]$. If $Sig[PSP] = \Sigma$, we refer to PSP as a $\Sigma-$specification.

Consider an arbitrary but fixed general logic,

Definition 7. *For every signature Σ in* **SIGN** *we define $\Sigma-$viewpoints PSP by*

- any $\Sigma-$*presentation* $PRES = (\Sigma, Eq)$ is a viewpoint PSP with the following semantics

$Sig[PRES] = \Sigma$
$Sem[PRES] = \{M \in |Sem(\Sigma)|, M \models Eq\} \stackrel{not}{=} Sem_{\Sigma}[Eq]$

- **enrichment** : $unif(PSP, PSP')$ of the $\Sigma-$viewpoint PSP with the $\Sigma'-$viewpoint PSP' , where $\Sigma \cap \Sigma' = \emptyset$, is a viewpoint with the following semantics

$Sig[unif(PSP, PSP')] = \Sigma \cup \Sigma'$
$Sem[unif(PSP, PSP')] = Sem[PSP] \cup Sem[PSP']$
A bit of syntactic sugar: we write the enrichment operation as
spec *ENRICHMENT* **is**
enrich *PSP_name* **with**
PSP′
endspec.

Example 8. *Consider the parallel specification of a telephone account by two different teams. The first team's perspective is that of an account in credit. It uses the partial logic $\mathcal{LTSHL}$ to define a general consume operation on the account. The second team concentrates on deposit operation on the account and uses the $\mathcal{FOL}^{=}$ logic.*

spec *CONS* **is**
enrich *AMOUNT* **with**
sort Sum
hidden sort State
const ok, *outm* : State;
opns $Consume : State\ Sum \nrightarrow State$
$Balance : State \rightarrow Sum$
var S : State; X : Sum;
axioms
$(S = ok) - \{\ Balance(S) \geq X \Rightarrow Balance(Consume(S, X)) = Balance(S) - X\} \rightarrow (S = ok)$.
$(S = ok) - \{\ Balance(S) < X \Rightarrow Balance(Consume(S, X)) = Balance(S) - X\} \rightarrow (S = outm)$.
endspec

The specification *AMOUNT* defines a standard numerical datatype like integers. Observe that the operation *Consume* is partial because is not defined in the state *outm*.

spec *DEP* **is**
enrich *AMOUNT, BOOLEAN* **with**
sort State, Amount
const init : State;
opns $Add : Credit\ Amount \rightarrow Credit$
$Acc\_Value : Credit \rightarrow Amount$
var S : Credit; X : Amount;
axioms
$Positive(init) = true$.
$(Positive(S) = true) \Rightarrow (Positive(Add(S, X)) = true) \wedge (Acc\_Value(Add(S, X)) = Acc\_Value(S) + X)$.
$(Positive(S) = false) \wedge (Acc\_Value(S) + X < 0) \Rightarrow (Positive(Add(S, X)) = false) \wedge (Acc\_Value(Add(S, X)) = Acc\_Value(S) + X)$.
$(Positive(S) = false) \wedge (Acc\_Value(S) + X \geq 0) \Rightarrow (Positive(Add(S, X)) = true) \wedge (Acc\_Value(Add(S, X)) = Acc\_Value(S) + X)$.
endspec

.

Definition 9. *(MC-**refinement**) A (partial) specification REF refines (by model containment MC) a viewpoint PSP, and we note this by* $REF \sqsupseteq PSP$ *if*

$$\begin{aligned} Sig[REF] &\subseteq Sig[PSP] \\ Sem[REF] &\subseteq Sem[PSP] \end{aligned}$$

We adopt the view on viewpoints consistency expressed in [6] by

> "A collection of viewpoints is consistent if and only if it is possible for at least one example of an implementation to exist that can conform to all viewpoints."

Definition 10. *(**unification**) ([6]) The unification* $UNIF[PSP, PSP']$ *of two viewpoints* PSP *and* PSP' *is defined as the smallest common refinement*

$UNIF[PSP, PSP'] \sqsupseteq PSP$
$UNIF[PSP, PSP'] \sqsupseteq PSP'$
$SP \sqsupseteq PSP', SP \sqsupseteq PSP \Rightarrow SP \sqsupseteq UNIF[PSP, PSP']$

An effective procedure for constructing the unification of axiomatic specified viewpoints can be found in our paper [7].

Example 11. *Consider the viewpoints DEP and CONS from the Example* ***8****. Their unification is*

spec *UNIF* **is**
enrich *AMOUNT, BOOLEAN* **with**
sort Amount
hsort State
const ok, *outm* : State;
opns $Consume : State\ Amount \nrightarrow State$
$Add : State\ Amount \rightarrow State$
$Balance : State \rightarrow Amount$
var S : State; X : Amount;
axioms
$(S = ok) - \{\ Balance(Add(S, X)) = Balance(S) + X\ \} \rightarrow (Add(S, X)) = ok).$
$(S = outm)\ -\{Balance(S) + X < 0 \Rightarrow Balance(Add(S, X)) = Balance(S) + X\ \} \rightarrow (Add(S, X)) = outm).$
$(S = outm)\ -\{\ Balance(S) + X > 0 \Rightarrow Balance(Add(S, X)) = Balance(S) + X\ \} \rightarrow (Add(S, X)) = ok).$
$(S = ok)\ -\{\ Balance(S) \geq X \Rightarrow Balance(Consume(S, X)) = Balance(S) - X\ \} \rightarrow (Consume(S, X)) = ok).$
$(S = ok)\ -\{\ Balance(S) < X \Rightarrow Balance(Consume(S, X)) = Balance(S) - X\ \} \rightarrow (Consume(S, X)) = outm).$
endspec
Of course, the concrete method (pushout applied to some translations of the viewpoints) for construction of the unification makes necessary supplementary constructions, which will be given in the rest of the paper. In particular, the following signature morphism has been applied to *DEP*
$Credit \rightarrow State$, $Acc\_Value \rightarrow Balance$, $true \rightarrow ok$, $false \rightarrow outm$, $Positive \rightarrow Current$

3. A FRAMEWORK FOR COMPOSITE VIEWPOINTS

3.1. Translation Between Specification Logics. Over the last ten years there have been a lot of variations in defining morphisms of logics. These notions have been given many different names, including morphism, map, embedding, simulation, transformation, coding, representation, and more, most of which do little or nothing to suggest their nature. We present in this paragraph what are, in our view, the most useful and logically articulated definitions..

Let $\mathcal{L} = (\mathbf{SIGN}, Syn, Sem, \models, \vdash)$ and $\mathcal{L}' = (\mathbf{SIGN}', Syn', Sem', \models', \vdash')$ be two categorical specification logics.

Definition 12. *A logic morphism (Goguen [14]) Mor between logics*

$$\partial = (\phi, \alpha, \beta) : \mathcal{L} \rightarrow \mathcal{L}'$$

is given by

- a functor $\phi : \mathbf{SIGN} \rightarrow \mathbf{SIGN}'$
- a natural transformation $\alpha : \phi; Syn' \Rightarrow Syn : \mathbf{SIGN} \rightarrow \mathbf{SET}$
- a natural transformation $\beta : Sem \Rightarrow \phi; Sem' : \mathbf{SIGN}^{op} \rightarrow \mathbf{CAT}$ such that the simple map of institutions condition

$\beta(\Sigma)(M) \models'_{\phi(\Sigma)} f' \iff M \models_\Sigma \alpha(\Sigma)(f')$ (*Satisfaction Condition*)
holds for each $\Sigma \in |\mathbf{SIGN}|$, $M \in |Sem(\Sigma)|$, $f' \in Syn'(\phi(\Sigma))$.

The functor ϕ on signatures and the natural transformation β on models go in the same direction in this definition, while the natural transformation α goes in the opposite direction.

Logic morphisms compose and form a category **LOG**.

Definition 13. *A logic comorphism $CMor$ (or a plain map -Meseguer [22]) between logics*

$$\partial^{op} = (\phi, \alpha, \beta) : \mathcal{L} \to \mathcal{L}'$$

is given by

- a functor $\phi : \mathbf{SIGN} \to \mathbf{SIGN}'$
- a natural transformation $\alpha : Syn \Rightarrow \phi; Syn' : \mathbf{SIGN} \to \mathbf{SET}$
- a natural transformation $\beta : \phi; Sem' \Rightarrow Sem : \mathbf{SIGN}^{op} \to \mathbf{CAT}$ such that the simple map of institutions condition

$\beta(\Sigma)(M') \models_\Sigma f \iff M' \models_{\phi(\Sigma)} \alpha(\Sigma)(f)$ (Co-*Satisfaction Condition*)
holds for each $\Sigma \in |\mathbf{SIGN}|$, $M \in |Sem(\Sigma)|$, $f' \in Syn'(\phi(\Sigma))$.

Logic comorphisms compose and form a category **COLOG**.

Proposition 14. **LOG** *and* **COLOG** *are both complete.*

Because we are primarily interested in specification morphisms, and in our approach specifications are theories, we need to consider generalizations of logic morphisms that involve mapping theories instead of just signatures.

Let $Sg : \mathbf{TH} \to \mathbf{SIGN}$ be the functor which forgets the formulas of a theory.

Definition 15. *The logic of specifications over a general logic $\mathcal{L} = (\mathbf{SIGN}, Syn, Sem, \models, \vdash)$ is $\mathcal{L}^{SP} = (\mathbf{TH}^{\mathcal{L}}, Syn^{SP}, Sem^{SP}, \models^{SP}, \vdash^{SP})$ where $\mathbf{TH}^{\mathcal{L}}$ is the category of theories over $\mathcal{L}$, Syn^{SP} is $Sg; Syn$, Sem^{SP} is the extension of Sem to theories, $\models^{SP}$ is $Sg; \models$.*

Definition 16. *A specifications morphism is a morphism from $\mathcal{L}^{SP}$ to $\mathcal{L}'^{SP}$ which is signature preserving.*

We denote by **SPLOG** the category of logics of specifications and their morphisms.

In the rest of this paper we will use the notation **LOGICS** to denote anyone of the categories **LOG**, **COLOG** and **SPLOG**.

Example 17. *(A translation between $\mathcal{PFOL}^{=}$ and $\mathcal{LTSHL}$) Consider the specification logics presented Examples 5 and 6.*

$\partial(Credit) = State$, $\partial(Sum) = Amount$,
$\partial(Consume) = (Consume, \{a1, a2, a3\})$
$\partial(x) = x$ otherwise

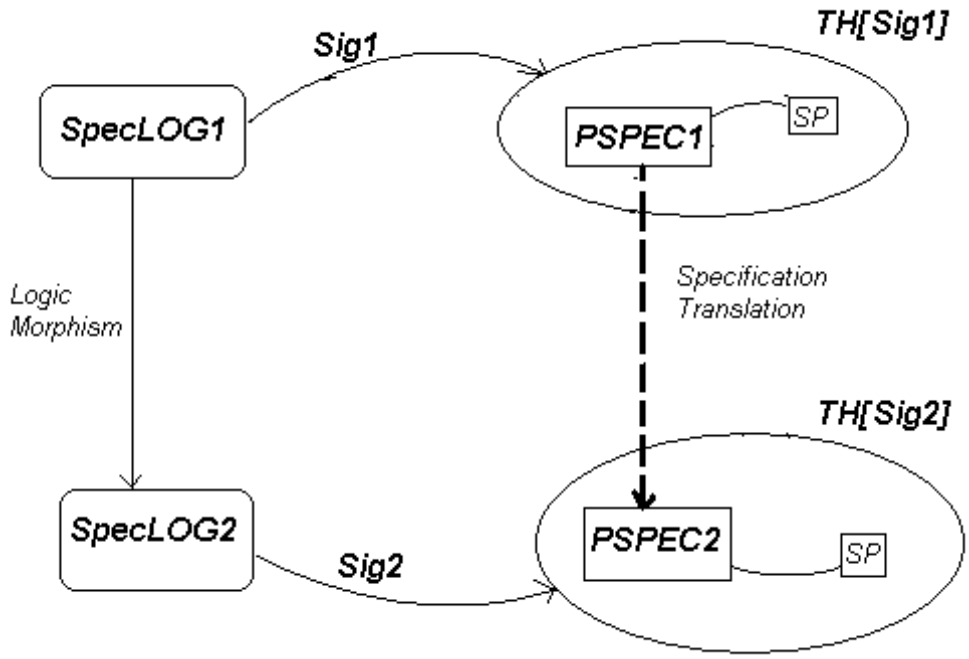


Figure 1:

3.2. A Category for Composite Viewpoints.

Definition 18. *Given a functor* $G : \mathbf{C}^{op} \rightarrow \mathbf{CAT}$, *the Grothendieck construction constructs the fibration induced by* G, *i.e. a category* $\mathbf{F}(\mathbf{C}, G)$ *defined as follows:*

- an object of $(\mathbf{C}, G)$ is a pair $< a, x >$, where $a \in |\mathbf{C}|$ and $x \in |G(a)|$
- an arrow $< \alpha, \rho >:< a, x > \rightarrow < a', x' >$ has $\alpha : a \rightarrow a'$ an arrow of $\mathbf{C}$ and $\rho : x \rightarrow G\alpha(x')$ an arrow of $F(a)$
- if $< \alpha, \rho >:< a, x > \rightarrow < a', x' >$ and $< \beta, \sigma >:< a', x' > \rightarrow < a'', x'' >$ then $< \alpha, \rho > ; < \beta, \sigma >:< a, x > \rightarrow < a'', x'' >$ is defined as$< \alpha, \rho >; < \beta, \sigma > = < \alpha; \beta, \rho; G\alpha(\rho) >$

This construction provides us with a way to give a domain where objects are very general logical structures and whose arrows are compatible relations between them.

Let $\partial = (\phi, \alpha, \beta) : \mathcal{L} \rightarrow \mathcal{L}'$ be a (co)morphism of logics and $< \Sigma, \Gamma >$, $< \Sigma', \Gamma' >$ theories from **TH** and **TH**$'$.

Definition 19. *A* $\partial-$*composite viewpoint translation between* $< \Sigma, \Gamma >$ *and* $< \Sigma', \Gamma' >$ *is an arrow* $\clubsuit :< \Sigma, \Gamma > \rightarrow < \Sigma', \Gamma' >$ *such that* $\clubsuit : \phi(\Sigma) \rightarrow \Sigma'$ *is a morphism in* **SIGN**$'$ *and such that* $\Gamma' \vdash_{\Sigma'} Syn(\phi(\Sigma))(\alpha(\Sigma)(\Gamma) \cup \emptyset'_{\Sigma})$.

The *composite viewpoint translation*s compose and build a category which is on top of the category **LOGICS**.

Theories from arbitrary logics as objects and *composite viewpoint translation*s as morphisms define the category **HPSPEC**.

The category **HPSPEC** is exactly the Grothendieck category $G(\mathbf{LOGICS}, Th)$ where $Th : \mathbf{LOGICS} \rightarrow \mathbf{CAT}$.

Given a *composite viewpoint translation* $\clubsuit :< \Sigma, \Gamma > \rightarrow < \Sigma', \Gamma' >$, its denotation is the reindexing functor $Sem(\clubsuit) : Sem'_{\models}(< \Sigma', \Gamma' >) \rightarrow Sem_{\models}(< \Sigma, \Gamma >)$, which maps models from the target to the source logic.

Definition 20. *(Indexed Category of Composite Models) Let $G(LOGICS, Th)$ be the Grothendieck category. and $< \Sigma, \Gamma >$, $< \Sigma', \Gamma' >$ theories from $\mathbf{TH}_{\mathcal{L}}$ and $\mathbf{TH}'_{\mathcal{L}'}$. Then the assignment $< \Sigma, \Gamma > \mapsto Sem_{\models,\mathcal{L}}(< \Sigma, \Gamma >)$,*

$\clubsuit :< \Sigma, \Gamma > \rightarrow < \Sigma', \Gamma' > \rightarrow Sem(\clubsuit) : Sem'_{\models,\mathcal{L}}(< \Sigma', \Gamma' >) \rightarrow Sem_{\models,\mathcal{L}}(< \Sigma, \Gamma >)$ defines an indexed functor $HSem : G(\mathbf{LOGICS}, Th)^{op} \rightarrow \mathbf{CAT}$.

Definition 21. *We define the category of composite models and composite morphisms as the split fibration induced by $HSem : G(\mathbf{LOGICS}, Th)^{op} \rightarrow \mathbf{CAT}$, i.e. the category $\mathbf{F}(G(\mathbf{LOGICS}, Th), HSem)$.*

In general the indexed category of composite models and composite model functors $HSem : G(\mathbf{LOGICS}, Th)^{op} \rightarrow \mathbf{CAT}$ as the semantically functor which maps composite viewpoints to their model theoretical counterparts.

3.3. A Logic for Composite Viewpoints. A general logic for specifying the unification must be able to keep some original viewpoint specification logics. A candidate logics for this are the Grothendieck logics.

Definition 22. *(Diaconescu [10]) Given an indexed logic $\mathcal{L} : Ind^{op} \rightarrow \underline{Log}$, define the Grothendieck logic $\mathcal{L}^{\#}$ as follows:*

-Signatures in $\mathcal{L}^{\#}$ are pairs $(\sum, i)$, where $i \in |Ind|$ and $\sum$ a signature in the logic $\mathcal{L}(i)$,
-signature morphisms $(\sigma, d) : (\sum_1, i) \rightarrow (\sum_2, j)$ consist of a morphism $d : i \rightarrow j \in Ind$ and a signature morphism $\sigma : \sum_1 \rightarrow \Phi^{\mathcal{L}(d)}(\sum_2)$ (here, $\mathcal{L}(d) : \mathcal{L}(j) \rightarrow \mathcal{L}(i)$ is the logic morphism corresponding to the arrow $d : i \rightarrow j$ in the logic graph, and $\Phi^{\mathcal{L}(d)}$ is its signature translation component),
-the $(\sum, i)-$sentences are the $\sum$-sentences in $\mathcal{L}(i)$, and sentence translation along (σ, d) is the composition of sentence translation along σ with sentence translation along $\mathcal{L}(d)$,
-the $(\sum, i)-$models are the $\sum-$models in $\mathcal{L}(i)$, and model reduction along (σ, d) is the composition of model translation along $\mathcal{L}(d)$ with model translation along σ, and
-satisfaction (resp. entailment) w.r.t. $(\sum, i)$ is satisfaction (resp. entailment) w.r.t. $\sum$ in $\mathcal{L}(i)$.

4. Formal Testing for Axiomatic Specifications

We develop our formal approach to software testing in the algebraic tradition initiated by M.C. Gaudel and B. Marre in [2] and [21].

A *testing hypothesis* describes assumptions about the system or the test process to reduce the size of the test set. But too strong hypotheses could weaken the generated tests, as an industrial experiment [21] shows. We consider here only *uniformity hypotheses*: we define sub-domains of interpretation for items, such that the system has the same behavior for all the values of the sub-domain. Then we assume that testing for a value of a sub-domain is enough to test for all its values. The sub-domains are defined in a logical form by the concept of *test cases*.

Definition 23. *A test instance is a couple (C, D) where C is a test case corresponding to a test hypothesis and D is a test data defined with respect to this test hypothesis.*

Let $Prog$ be the program under test and a a specific test goal (like a axiom of the program specification or a desirable program property). If an execution of $Prog$ satisfies one $\Sigma-$test data set D , we say "$Prog$ satisfies the test data set D" and we notice $Prog \ddot{\sim}_\Sigma D$ and $Prog \ddot{\sim}_\Sigma C$. We use the same notation for a test case C: $Prog \ddot{\sim}_\Sigma C$ means that all the executions of the program $Prog$ satisfies C (and we say "P satisfies the test case C").

Definition 24. *A test instance (C, D), with respect to a program $Prog$, is called*

- *unbiased* if $(Prog \ddot{\sim} C) \wedge (Prog \models_\Sigma a) \Rightarrow Prog \ddot{\sim}_\Sigma D$.
- *valid* if $(Prog \ddot{\sim} C) \wedge (Prog \ddot{\sim}_\Sigma D) \Rightarrow Prog \models_\Sigma a$.
- *correct* if it is valid and unbiased: $(Prog \ddot{\sim} C) \Rightarrow (Prog \models_\Sigma a \Longleftrightarrow Prog \ddot{\sim}_\Sigma D))$

Unbiased test sets are ones which accept all correct programs, but incorrect programs can also be accepted as correct. Valid tests do not accept incorrect programs, but correct programs can be rejected. Thus, an ideal test must be valid and unbiased, i.e. correct.

Definition 25. *The category of $\Sigma-$test cases, denoted $\mathbf{TC}_\Sigma$, has*

- *objects: test cases*
- *morphisms: for a test case morphism $C \rightarrow C'$, at each instance of C we associate a an instance of C' with the same item, i.e. an inclusion between the set of instances of both test cases (or equivalently the logical implication $C \Rightarrow C'$ holds).*

$\mathbf{TC}_\Sigma$ is a finitely cocomplete category: initial object is a an empty test case, pushout is defined by putting together the instances of both test cases without repetition of shared instances.

The instantiation of test cases with values generates the *test data sets*.

Definition 26. *A **test data set** is a subset of consistent ground instances of formulas from $\mathbf{TC}_\Sigma$.*

Example 27. *Consider the viewpoint DEP from the Example **8**. As a test cases we could consider an uniformity hypothesis like*

$$\begin{aligned}
C1 &= \{Positive(S) = true, Acc_Value(S) > 0, X > 0, Acc_Value(S) > X\} \\
C2 &= \{Positive(S) = true, Acc_Value(S) > 0, X = 0\} \\
C3 &= \{Positive(S) = true, Acc_Value(S) = 0, X = 0\} \\
C4 &= \{Positive(S) = false, Acc_Value(S) < 0, X > 0, Acc_Value(S) + X < 0\} \\
C5 &= \{Positive(S) = false, Acc_Value(S) < 0, X = 0\} \\
C6 &= \{Positive(S) = false, Acc_Value(S) < 0, X > 0, Acc_Value(S) + X > 0\} \\
C7 &= \{Positive(S) = false, Acc_Value(S) < 0, X > 0, Acc_Value(S) + X = 0\}
\end{aligned}$$

We can observe that from C one can derive

$$Positive(S) = true \Leftrightarrow Acc\_Value(S) \geq 0$$

Using a test notation like

$$< Positive(S), Acc\_Value(S), X, Positive(Add(S, X)), Acc\_Value(Add(S, X)) >$$

a test data set is

D_{DEP} ={ $< true, 25, 20, true, 45 >$,
$< true, 25, 0, true, 25 >$,
$< true, 0, 0, true, 0 >$,
$< false, -25, 20, false, -5 >$,
$< false, -25, 0, false, -25 >$,
$< false, -20, 25, true, 5 >$,
$< false, -25, 25, true, 0 >$ }
.

Example 28. *Consider the viewpoint CONS from the Example **8**. As a test cases we could consider an uniformity hypothesis like*

$$\begin{aligned} C1 &= \{Balance(S) > 0, X > 0, Balance(S) > X\} \\ C2 &= \{Balance(S) > 0, X = 0\} \\ C3 &= \{Balance(S) > 0, X > 0, Balance(S) = X\} \\ C4 &= \{Balance(S) = 0, X = 0\} \\ C5 &= \{Balance(S) > 0, X > 0, Balance(S) < X\} \\ C6 &= \{Balance(S) = 0, X > 0\} \end{aligned}$$

Using a test notation like

$$< Current(S), Balance(S), X, Consume(S, X), Balance(Consume(S, X)) >$$

a test data set is

D_{WITHDR} ={ $< ok, 25, 20, ok, 5 >$,
$< ok, 25, 0, ok, 25 >$,
$< ok, 25, 25, ok, 0 >$,
$< ok, 0, 0, ok, 0 >$,
$< ok, 20, 25, outm, -5 >$,
$< ok, 0, 25, outm, -25 >$ }
.

Definition 29. *(Testing functor) We define a functor Test giving for every signature its category of $\Sigma-$tests*

$$\begin{aligned} Test &: \mathbf{SIGN} \rightarrow \mathbf{CAT} \\ Test(\Sigma) &= \mathbf{TC}_\Sigma \end{aligned}$$

such that *the satisfaction condition for tests*

$$M' \mathrel{\tilde{\models}}_{\Sigma'} Test(\sigma)(a) \Leftrightarrow Sem(\sigma)(M') \mathrel{\tilde{\models}}_{\Sigma} a$$

is satisfied.

In a categorical specification logic, a program is identified with a viewpoint PSP and its execution with the semantics of the specification $Sem[PSP]$. Thus, a test goal is a formula $a \in TH[PSP]$, and the satisfaction relations $Prog \mathrel{\tilde{\models}}_\Sigma D$ and $Prog \mathrel{\tilde{\models}}_\Sigma C$ are replaced by the test satisfaction relation for models $\mathrel{\tilde{\models}}_\Sigma \subseteq |Mod(\Sigma)| \times |Test(\Sigma)|$ for each $\Sigma \in |\mathbf{SIGN}|$

5. TESTING UNIFICATION BY UNIFICATION OF TESTS

Suppose we have generated the test instances $(C_{PSP}, D_{PSP'})$ from the viewpoints PSP, PSP', correct for the test goals X_{PSP} , $X_{PSP'}$ and similar for the specification CO, which expresses the correspondences between the viewpoints. It is important to note that we do not use any hypotheses about the way tests have been obtained. For example, D_{PSP} could be obtained from a correctness proof, as in [20], and $D_{PSP'}$ could be obtained by any method of generating test cases from a formal specification.

Consider the unification obtained by the following pushout of viewpoints,

$$\begin{array}{ccccc} & & CO & & \\ & m_1 \swarrow & & \searrow m_2 & \\ PSP & & & & PSP' \\ & m_1' \searrow & & \swarrow m_2' & \\ & & UNIF & & \end{array}$$

corresponding to the pushout of signatures

$$\begin{array}{ccccc} & & Sig[CO] & & \\ & \sigma_1 \swarrow & & \searrow \sigma_2 & \\ Sig[PSP] & & & & Sig[PSP'] \\ & \sigma_1' \searrow & & \swarrow \sigma_2' & \\ & & Sig[UNIF] & & \end{array}$$

We are interested in the way (C_{UNIF}, D_{UNIF}) for $UNIF$ can be obtained. In the correctness of the method.

Proposition 30. *Correctness of test instances is preserved by signature translation and composite translation morphisms.*

Proposition 31. *The test instance (C_{UNIF}, D_{UNIF}) obtained by pushout*

$$\begin{array}{ccccc} & & (C_{CO}, D_{CO}) & & \\ & t_1 \swarrow & & \searrow t_2 & \\ (C_{PSP}, D_{PSP}) & & & & (C_{PSP'}, D_{PSP'}) \\ & t_1' \searrow & & \swarrow t_2' & \\ & & (C_{UNIF}, D_{UNIF}) & & \end{array}$$

is correct for $Syn(m_1')(X_{PSP}) \wedge Syn(m_2')(X_{PSP'})$.

Proof. By renaming according to m_1' we have a test instance (C'_{PSP}, D'_{PSP}) defined on $\mathrm{Si}\,g[PSP]$ and correct for $Syn(m_1')(X_{PSP})$ and similar we get a test instance $(C'_{PSP'}, D'_{PSP'})$ defined on $\mathrm{Si}\,g[PSP']$ and correct for $Syn(m_2')(X_{PSP'})$.
We consider a model M' defined on $\mathrm{Si}\,g[UNIF]$ such that M' satisfies $C_{UNIF} = C'_{PSP} \wedge C'_{PSP'}$. M' satisfies C'_{PSP} and $C'_{PSP'}$ and correctness of (C'_{PSP}, D'_{PSP}) and $(C'_{PSP'}, D'_{PSP'})$ gives

$M' \models_{\mathrm{Si}\,g[UNIF]} Syn(m1)(X_{PSP}) \Longleftrightarrow M' \leftrightarrows_{\mathrm{Si}\,g[UNIF]} D'_{UNIF1}$
$M' \models_{\mathrm{Si}\,g[UNIF]} Syn(m2)(X_{PSP'}) \Longleftrightarrow M' \leftrightarrows_{\mathrm{Si}\,g[UNIF]} D'_{PSP'}$

If M' satisfies $D'_{PSP}$and $D'_{PSP'}$ we can build a consistent test case on $\mathrm{Si}\,g[UNIF]$ by conjunction, so M' satisfies D_{UNIF}. Reversely, if M' satisfies D_{UNIF},it exists a test case of D_{UNIF} satisfied by $M'$and build by composition of a test case of D'_{UNIF1} and a test case of $D'_{PSP'}$. So M' satisfies $D'_{PSP}$and $D'_{PSP'}$.
We obtained that

$M' \models_{\mathrm{Si}\,g[UNIF]} Syn(m1)(X_{PSP}) \wedge Syn(m2)(X_{PSP'}) \Longleftrightarrow M' \leftrightarrows_{\mathrm{Si}\,g[UNIF]} D_{UNIF}$

Because $X_{UNIF} = Syn(m1)(X_{PSP}) \wedge Syn(m2)(X_{PSP'})$ we obtain that the test instance (C_{UNIF}, D_{UNIF}) is correct for $Syn(m_1')(X_{PSP}) \wedge Syn(m_2')(X_{PSP'})$.q.e.d.

Example 32. *We construct now the test instance* (C_{UNIF}, D_{UNIF}) *from* (C_{WITHDR}, D_{WITHDR}) *and* (C_{DEP}, D_{DEP}).

We consider the following notation for test data

$$< op, Current(S), Balance(S), X, Current(op(S, X)), Balance(op(S, X)) >$$

The symbol *op* ranges in the values $\{Add, Consume\}$ and the symbol $\perp$ when the attribute is not defined for the current operation. The test data D_{UNIF} is

$D_{UNIF} =\{ < Consume, ok, 25, 20, ok, 5 >,$
$< Consume, ok, 25, 0, ok, 25 >,$
$< Consume, ok, 25, 25, ok, 0 >,$
$< Consume, ok, 0, 0, ok, 0 >,$
$< Consume, ok, 20, 25, outm, -5 >,$
$< Consume, ok, 0, 25, outm, -25 >,$
$< Add, ok, 25, 20, ok, 45 >,$
$< Add, ok, 25, 0, ok, 25 >,$
$< Add, ok, 0, 0, ok, 0 >,$
$< Add, outm, -25, 20, outm, -5 >,$
$< Add, outm, -25, 0, outm, -25 >,$
$< Add, outm, -20, 25, ok, 5 >,$
$< Add, outm, -25, 25, ok, 0 > \}.$

Similar we construct C_{UNIF}

$$\begin{aligned}
C1 &= \{Op, ok, Balance(S) > 0, X = 0\} \\
C2 &= \{Op, ok, Balance(S) = 0, X = 0\} \\
C3 &= \{Op, ok, Balance(S) > 0, X > 0, Balance(S) > X\} \\
C4 &= \{Consume, ok, Balance(S) > 0, X > 0, Balance(S) = X\} \\
C5 &= \{Consume, ok, Balance(S) > 0, X > 0, Balance(S) < X\} \\
C6 &= \{Consume, ok, Balance(S) = 0, X > 0\} \\
C7 &= \{Add, outm, Balance(S) < 0, X > 0, Balance(S) + X < 0\} \\
C8 &= \{Add, outm, Balance(S) < 0, X = 0\} \\
C9 &= \{Add, outm, Balance(S) < 0, X > 0, Balance(S) + X > 0\} \\
C10 &= \{Add, outm, Balance(S) < 0, X > 0, Balance(S) + X = 0\}
\end{aligned}$$

6. CONCLUSIONS

As the title suggests, this work is only preliminary. Much more remains to be done in the sense of using effective test case procedures and describing concrete development relations to be used in the unification process. We intend to instantiate our framework with more formal notations, like temporal logics and co-algebraic specifications.

The basic achievements of this work are:

- a method of unification of heterogeneous viewpoints using category theory,
- corresponding to this, a method of testing the unification by unifying the tests generated from the viewpoints
- an exemplification using a small case study (a simplified telephone account)
- a formal framework for constructing proofs of correctness preservation for the unification process.

The formal testing theory we have presented here is connected with general logics only on their model theoretic counterpart. Despite of a careful presentation of proof theory, issues like general logics for theorem provers, mappings of viewpoint logics into a universal logic didn't get their natural place now. A future work might describe all these issues in connection with using correctness proofs as a source for test case generation [20].

This work is all theoretical, conducted by a toy example. A next step is to consider an executable logical framework (like LF [15], Maude or HOL) in which the Grothendieck viewpoint logic should be embedded, and a real life case study (we target an air traffic control example).

As future 'test' study, we will concentrate on testing of ODP and UML system specifications. ODP defines a basic framework for conformance as part of the reference model, rather than it being retrofitted later, as in OSI. UML allows systems UML allows systems to be described using diagrams and notations of various kinds, but none of these is assumed to fully characterise the behaviour of the system being specified. Thus, for an UML specification, any diagram could be mapped into a viewpoint specification, and consequently develop a specific theory of testing. We also intend to apply our logical viewpoint specification and testing framework to hybrid systems. This can be done in few ways, for example by extending UML capabilities to specify hybrid systems or by modelling them using a particular categorical logic.

Acknowledgements

Authors want to thank Dr. Eerke Boiten who contributed to the first part of this paper (we have actually used his work on a categorical framework for viewpoint specifications) and read a draft of this paper. The financial support from EPSRC under the grant GR/N03389/01 is also fully acknowledged.

References

[1] J. Adamek, H. Herrlich, G. Strecker **Abstract and Concrete Categories** Wiley, New York, 1990.

[2] G Bernot, M–C Gaudel, B Marre *Software Testing Based on Formal Specifications: a Theory and a Tool* Software Engineering Journal, Nov 1991.

[3] J. Bicarregui, M. Bishop, T. Dimitrakos, K. Lano, T. Maibaum, B. Matthews, B. Ritchie *Supporting Co-Use of VDM and B by Translation* In J. Bicarregui, J. Fitzgerald (Eds) VDM in 2000! Proceedings of the 2nd VDM workshop, 2000.

[4] E.A. Boiten, J. Derrick *A Constructive Framework for Partial Specification* EPSRC Fast Stream Research Proposal Case for Support http://www.cs.ukc.ac.uk/research/tcs/framework/

[5] E.A. Boiten, H. Bowman, J. Derrick, P.F. Linington, M.W.A. Steen *Viewpoint Consistency in ODP*. Computer Networks, 34(3):503-537, 2000.

[6] H. Bowman, E. A. Boiten, J. Derrick, M. W. A. Steen. *Strategies for Consistency Checking Based on Unification*. Science of Computer Programming, 33:261-298, 1999.

[7] M.C. Bujorianu, E.A. Boiten *Consistency Checking and Unification of Partial Specifications Using Category Theory* submitted.

[8] M.C. Bujorianu, M.L. Bujorianu *On the Hilbert Machines Quantitative Computational Model* QAPL'01 Workshop on Quantitative Aspects of Programming Laguages, Satellite Event of PLI'02, September 3 - 7, 2001 – Firenze, Italy. To appear in Vol. 60 of Electronic Notes in Theretical Computer Science, Elsevier, www.elsevier.locate/tcs/

[9] M.C. Bujorianu, M.L. Bujorianu *Linear Logic: From Stochastic Analysis to Software Testing* BCTCS 18 British Colloquium for Theoretical Computer Science, Bristol, UK, 2002.

[10] R. Diaconescu. *Extra Theory Morphisms for Institutions: Logical Semantics for Multi-paradigm Languages*. Technical Report IS-RR-96-0024S, Japan Institute of Science and Technology, 1996.

[11] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, M. Goedicke. *VIEWPOINTS: a Framework for Integrating Multiple Perspectives in System Development*. International Journal on Software Engineering and Knowledge Engineering, 2(1):31-58, 1992.

[12] M.C. Gaudel, P.R. James *Testing Algebraic Data Types and Processes: A Unifying Theory*. Formal Aspects of Computing , 10(5-6), pp. 436-451, 1998.

[13] M.C. Gaudel *Testing Can Be Formal, Too*. TAPSOFT 1995, pp. 82-96, 1994.

[14] J. Goguen, G. Rosu *Institution Morphisms* to appear in Formal Aspects of Computing.

[15] T. Harper, R., Sannella, D., Tarlecki, A. *Structured Theory Presentations and Logic Representations*. Annals of Pure and Applied Logic 67:113–160 1994.

[16] P.F. Linington, J. Derrick, H. Bowman *The Specification and Conformance of ODP Systems*. In 9th International Workshop on Testing of Communicating Systems, IFIP TC6/WG6.1, Chapman & Hall, pp. 93-114. 1996.

[17] I. MacColl, D. Carrington *Testing MATIS: a Case Study on Specification-based Testing of Interactive Systems* Formal Aspects of Human Computer Interaction Workshop, Sheffield, UK September 1998

[18] P. D. L. Machado. *On Oracles for Interpreting Test Results Against Algebraic Specifications*. In Proceedings of AMAST'98, volume 1548 of LNCS, Springer, 1999.

[19] S. Maharaj *Towards a Method of Test Case Extraction from Correctness Proofs*. in Proceedings of the 14th International Workshop on Algebraic Development Techniques, Bonas, France, pp 45-46, November 1999.

[20] S. Maharaj *Test Case Extraction from Correctness Proofs Case for Support* 2000 http://www.cs.stir.ac.uk/~sma/testing.ps

[21] B. Marre, P. Thévenod-Fosse, H. Waeselynck, P. Le Gall et Y. Crouzet. *An Experimental Evaluation of Formal Testing and Statistical Testing, IV. Fault Removal*. Section C., pages 273–281. ESPRIT Basic Research Series, Predictably Dependable Computing Systems. Springer-Verlag, 1995. Also SAFECOMP'92, Randell, B. and Laprie, J.-C. and Kopetz, H. and Littlewood, B. (ed.)

[22] J. Meseguer *General Logics*. In Logic Colloquium'87, pp. 275–329, North Holland, 1989.

[23] D. Pemberton, I. Sommerville *VOCAL: A Framework For Test Identification & Deployment* in IEE Proceedings - Software Engineering, Vol. 144, Issue 5-6, , p. 249 - 260, 1997.

[24] V. Wiels, S. Easterbrook *Management of Evolving Specifications Using Category Theory* Automated Software Engineering 98 October 13-16 1998, Hawai