A flexible environment to evaluate state–based test techniques

R. M. Hierons
Department of Information Systems and Computing
Brunel University, Uxbridge
Middlesex, UB8 3PH, UK
rob.hierons@brunel.ac.uk

ABSTRACT
In this position paper we argue that the presence of a flexible test environment, that allows the rapid prototyping of test techniques, would facilitate empirical research in software testing. Such an environment could be combined with a set of benchmark systems and specifications in order to allow researchers to rapidly prototype and evaluate new techniques. In this paper we focus on some of the requirements for a description language to be used by such an environment.

Categories and Subject Descriptors
D2.4 [Software Engineering]: Software/Program Verification; D2.2 [Software Engineering]: Testing and Debugging

1. INTRODUCTION
Testing is usually based around satisfying some test criterion. There are currently many standard test criteria, such as the test suite covering every transition [8], the suite set being a checking experiment [8], and the test suite providing 100% branch coverage (see, for example, [3, 4]). Despite the interest in criteria and techniques, there still remain significant doubts regarding how good the criteria and techniques are. While there has been some recent progress in the theory of comparing test techniques and criteria [7, 9], these doubts will only be resolved by sufficient experimentation using real case studies from a range of domains. This paper focuses on the testing of an important class of systems: state–based systems.

The following points suggest that in order to evaluate a test technique on real case studies it is important to implement the technique in the form of a tool.

1. Some large, complex case studies should be used since otherwise it is be difficult to investigate issues regarding scale. This is unlikely to be feasible without tool support.

2. The involvement of the experimenter, or even other humans, in test generation can introduce bias. Automation limits such bias.

The requirement to implement any new test technique, in order to effectively evaluate it, makes experimentation expensive.

There are many different application domains and systems within the different domains can have very different properties. Thus, since the efficacy of criteria and techniques depends crucially upon properties of the system under test (SUT), general results may only be obtained through evaluation across a range of application domains. It is thus important that any test environment developed can be applied across a number of problem domains.

This paper proposes the development of an environment for the rapid prototyping of test generation tools for the testing of state–based systems. Such an environment, combined with suitable benchmark case studies, would allow researchers to rapidly prototype and evaluate new test criteria/purposes and techniques. The rest of this paper briefly outlines the role of a test environment and elements that must be developed or incorporated within such an environment. It then focuses on the need for an appropriate description language.

2. THE ROLE OF A TEST ENVIRONMENT
A test environment should allow the user to specify test purposes and solution strategies, used to satisfy a given test purpose, for the SUT. Test generation can then be seen as a process of test purpose refinement: we start with a test purpose, such as satisfying a given criterion or applying a particular test generation technique. The test purpose is refined through a number of steps finally leading to a test suite that satisfies the initial test purpose. It is important that at each step the new test purpose $P'$ ‘conforms’ to the previous test purpose $P$: any test suite that satisfies $P'$ is guaranteed to satisfy $P$.

Test purposes and techniques may take account of the specification and/or the code. They may also utilize additional information about the SUT, possibly described using test
hypotheses [5] or a fault model [8], provided by the tester. Thus, a test environment must be able to access and manipulate such entities. Further, specifications and programs may be written in any one of a range of languages from several paradigms. In order to be useful a test environment must be able to handle the most common paradigms and languages.

The test environment will be used to implement and evaluate a given test purpose or technique. Test techniques often incorporate common solution mechanisms such as: constraint solvers; model checkers; standard graph and network algorithms; metaheuristic algorithms; and random generators. For example, a test technique might say: produce a set of subsequences that, between them, test the transitions and combine these optimally using a particular algorithm for solving the rural Chinese postman problem (see, for example, [2]). Thus, in order to allow test techniques to be rapidly prototyped, a test environment must have access to solution mechanisms and must be able to integrate these in order to solve a problem. The user must also be able to specify the application of such a mechanism within their test technique. Further, the user should be able to add, and refer to, additional solution mechanisms.

It is clear that a general test environment must be able to describe, access, and manipulate a wide range of entities at different levels of abstraction. Further, it must be able to use a wide range of solution mechanisms. This suggests the need for a single description language. Such a language must be extremely general. A test environment must also incorporate tools to translate between the languages actually used and this description language. The translation process is likely to be non-trivial, especially since users expect to receive information from the environment, regarding the specification and code, using the languages in which these were written: they will not want responses in the description language. We will now outline some of the requirements of such a description language.

3. PROPERTIES OF A DESCRIPTION LANGUAGE

The test environment must be able to manage programs, specifications, and test criteria. It must also be able to use a range of solution mechanisms. Further, the user must be able to describe their proposed test purpose or technique in a manner that the environment may use. This suggests that there must be a description language\(^1\) at the heart of the environment. The development of such a description language, that is applicable across a range of application domains, will form a substantial research project in itself.

In order to be sufficiently general, a description language must include at least the following aspects.

1. For the specification:
   (a) A logical state structure;
   (b) Data, in the form of variables;
   (c) Preconditions/guards and operations;
   (d) Real–time aspects; and
   (e) Concurrency and hierarchy: there must be several forms of each, possibly achieved using parameters that define the options for the semantics. For example, there are many different semantics for statecharts.

2. For the SUT:
   (a) Syntax and semantics that allow the representation of programs in common imperative and object-oriented languages;
   (b) Test hypotheses or a fault model that allow the tester to place constraints on the SUT. Such constraints represent properties the user believes the SUT has or are used to focus attention on certain classes of potential faults or failures.

3. Aspects that allow the user to describe test purposes and techniques. These should include at least the following:
   (a) Standard test requirements/purposes (such as include every transition, satisfy 100% branch coverage);
   (b) Ways of referring to standard solution mechanisms, such as model checkers and constraint solvers, to be used in order to satisfy the requirements;
   (c) Constructive test purposes, such as find the shortest path to a state using a particular algorithm;
   (d) Access to random generators;
   (e) An extendability that allows the user to interface the environment with additional tools and to ‘call’ these tools as part of a test purpose or technique.

Note that it is also necessary to have translators, between the description language and common specification, programming, and tool languages in order to facilitate generality and integration.

The description language will allow test requirements to be expressed at a number of degrees of abstraction. It will include statements that represent test purposes, such as executing every transition, or common test requirement patterns. These high–level test descriptions will be converted into test suites by tools that refine the test purposes, a test purpose potentially naming the refinement mechanism to be used. Specifications may also be written at very different levels of abstraction (compare, for example, statecharts [6] and Z [10] or B [1]). It would be useful if the language allowed a specification or test purpose to use declarative statements, possibly written in predicate logic.

4. CONCLUSIONS

This paper has proposed the development of a flexible test environment that allows the rapid prototyping and evaluation of test techniques. When combined with benchmark systems and specifications, such an environment would allow researchers to rapidly evaluate new test techniques on a range of systems from several application domains.
A general environment must be able to refer to specifications and programs written in a range of languages from several paradigms. This requires the development of a description language and translators between the common specification, programming, and tool languages and this description language. It is important that the environment also allows the user to provide information about the system under test (SUT) in the form of test hypotheses or a fault model. The environment must also be able to access the test purpose or technique and thus it seems natural to allow these to be stated in the description language. Finally, the user should be able to add solution mechanisms and be able to refer to these within their test techniques: this requires the description language to be extensible.

Given such a description language, test suite generation may be seen as a process of refinement. We start with a test purpose (which might be a test criterion or a test technique), a specification, and code. Through a sequence of (refinement) steps we take the test purpose and make it more concrete. Each refinement step must be sound in that if we satisfy the new entity then the resultant test suite is guaranteed to satisfy the previous entity. The process ends when we have a test suite.

Many research challenges remain. This paper has outlined some of the requirements for a general description language but much more work is required here. It is vital that a description language is sufficiently general but it is also desirable to restrict the language as far as possible in order to simplify the development of the environment. The specification of a description language is a major research challenge. There is a need to integrate commonly used tools into such an environment. In addition, the environment will require translators between the description language and commonly used specification and programming languages. Such translators must allow information returned to the user to be expressed in terms of the language they used, not the internal description language.

Acknowledgments
This work was partly supported by EPSRC grant FORTEST (Formal Methods and Testing). Many of the ideas described in this paper came out of discussions with Dr John Clark of York University, UK.

5. REFERENCES