Component Certification

John Morris, Gareth Lee, Kris Parker, Gary A Bundell[a], Peng Lam[b]

Abstract— **Component-Based Software Engineering relies on sources of reliable components: developers need to be able to trust the components with which they build larger systems. This has led to suggestions[1] that software certification laboratories should undertake the role of checking reliability of components offered for sale. This paper argues that is it more effective for developers to supply the test certificates in a standard, portable form, allowing purchasers to examine and run the test sets themselves and thus to form an opinion as to whether the component will meet their needs or not. By removing necessarily expensive third-party certification exercises, one of the major potential economic benefits of CBSE – the ability to build a complex system from inexpensive, but trusted, components will be retained.**

I. Introduction

Noting that most methods for certifying the quality of software products are process-based – requiring software publishers to 'take oaths concerning which development standards and processes they will use', Voas[1] has suggested that independent agencies – software certification laboratories (SCLs) – should take on a *product* certification role. He believes that 'completely independent product certification offers the only approach that consumers can trust'. In Voas' scheme, SCLs would

1. accept instrumented software from developers,
2. pass the instrumented software on to pre-qualified users,
3. gather information from user sites,
4. generate statistics on usage and performance in the field using the data gathered from several sites and
5. provide limited warranties for the software based on these statistics.

Additionally, the SCLs could, by continuing to collect data over time, broaden the warranty as the operational profile of the software broadened.

A. Limitations

We have identified several limitations of the SCL approach:

A.1 Cost

Of necessity, SCLs will have significant costs which will add to the cost of the certified product. A significant part of that cost will be insurance – necessary if the certifiers are to provide any form of guarantee to purchasers. The only benefit to a purchaser here is that insurers may be prepared – presumably after some some sufficiently long time interval – to factor the amount of operational data available into their calculation of risk and therefore premium.

A.2 Liability

To be effective, third party certifiers will need to provide some form of precisely stated warranty: purchasers can not be expected to pay a premium for their services without additional value. Whilst testing a component certainly adds value (*cf.* section IV), it is doubtful whether testing alone – without some form of guarantee of the completeness of it – will add sufficient value to make a SCL viable. User-based testing is essentially random testing (albeit biased to some 'operational profile') and thus exposes an SCL providing a warranty to significant damage claims from 'time-bombs' embedded in code which have never been exercised. It may be that SCLs are in an even more invidious position than the developers themselves. Developers simply disclaim liability: SCLs are providing professional advice to clients on the risk of using a component. As Voas notes, courts have not been kind when such advice has been proven faulty.

A.3 Developer Resources

Much successful, widely used software is written by a single programmer or small groups of programmers working independently (*e.g.* Linux, gcc, *etc.*). Their products would be expected to become a vital part of any thriving component market. However, since these individuals or groups are generally self-financed, it is unreasonable to expect that many of them would

- have the funds initially to pay for SCL services,
- be prepared to give away sufficient share of their efforts in the early stages to attract capital to pay for SCL services,
- have the time to invest in the negotiations with an SCL *or*
- be prepared to instrument their products for residual testing[2] if they were employing other formal methods for testing.

A.4 Safety-critical Systems

Voas[1] himself notes this as an area of special challenge; he recognizes that SCLs will have some difficulty persuading testers to fly software-controlled aircraft or submit to software-controlled medical devices. His solution is to attempt to certify the software in noncritical environments first: he suggests that once 'noncritical certification' is achieved, a product could be used with confidence in safety-

[a]Centre for Intelligent Information Processing Systems, Department of Electrical and Electronic Engineering, The University of Western Australia, Nedlands WA 6907, Australia email: [bundell,gareth,morris,kaypy]@ee.uwa.edu.au
[b]School of Engineering, Murdoch University, Murdoch WA 6150, Australia email: peng@eng.murdoch.edu.au

critical applications. There are a number of manifest problems here:

- firstly, many key components of safety-critical systems will have no application in non-critical systems and,
- secondly, the operational profiles collected from users are unlikely to satisfy the software product standards that are used in this area; for example, they would not be expected to satisfy coverage criteria required for airborne[3] or defence systems[4]. Thus testing to satisfy the applicable standard will still be needed. Furthermore, it is likely to consume the major part of any testing budget as even ensuring statement coverage requires test cases for many rare situations (unlikely to be covered in any reasonable period of actual use) to be constructed.

This paper proposes an entirely different model for software component certification, based on test certificates supplied by developers in a standard portable form so that purchasers may, in very short order, determine the quality and suitability of purchased software. Note that in the context of this paper – and for our test specifications – 'component' refers to any piece of software *with a well-defined interface* and thus encompasses components satisfying any of the plethora of definitions of the term found in the literature[5] as well as simple functions or procedures, such as those found in a mathematical library.

Some early proponents of component software based their ideas on the successful integrated circuit market – even referring to components as 'software ICs'. When describing an IC's capabilities, manufacturers have tended to adopt a fairly standard format: most data sheets are structurally similar – divided into DC and AC characteristics. The AC sections report propagation delays with similar notations and structures. This *de facto* standardisation is naturally of great benefit to design engineers: the standard structure of data sheets from different manufacturers makes them generally easy and fast to comprehend.

Our model clearly targets small components and does not necessarily imply that the SCL model has no place: firstly, there are many situations where the cost of a software failure is high and third party certifications by specialist testing organizations be the only acceptable model and secondly, SCL certification may be the only practical alternative for large software systems.

II. Developer Generated Tests

A. Standard Test Specifications

If developers are to supply test sets to purchasers, we need a standard, portable way of specifying tests, so that a component user may assess the reliability – or, alternatively, the degree of risk associated with use – of a component on an arbitrary target system.

We designed a test specification with the following aims:

1. it should be standard and portable,
2. it should be simple and easy to learn,
3. it should avoid language-specific features,

```
<TestSpecification>
  <TestSet Name="...">+
    <TestGroup Name="...">*
      <TestGroup>*
      <Invariant DataType="...">*
      <Operation Name="..." Pre="op_name ">*
        <Invariant>*
        <Constructor>*
        <MethodCall Target="...">*
    <Operation>
    <Invariant>
```

Fig. 1. Top Level Structure of a Test Specification. Only the basic structure is shown: some details have been omitted for clarity: the full DTD is in App A. Conventional multiplicity symbols are used: ? = optional; * = 0 or more; + = 1 or more.

```
<MethodCall Name="..." Target="..." Static="Y/N">
  <Arg Name="..." Source="..." DataType="...">*
  <Result Name="..." DataType="...">*
    <Exp>?
  <Exception Name="..." DataType="...">*
    <Exp>?
```

Fig. 2. Test Specification – `<MethodCall>` element: `<Constructor>` and `<MethodCall>` elements are essentially the same: refer to the DTD in App A for details.

4. it should work equally well with object oriented systems, simple functions and complex components (*e.g.* distributed objects, Java Beans, *etc.*),
5. it should efficiently handle the repetitive nature of many test sets,
6. tools to generate tests should be widely available and easily produced, *i.e.* it should not require proprietary software to generate test specifications,
7. it should not require proprietary software to interpret and run the tests *and*
8. it should support regression testing.

Our specification is depicted in tree form in figure 1 and formally described by the document type definition (DTD) set out in appendix A.

XML[6] enables us to meet requirement 1:

1. it has a standard developed by an independent organisation responsible for a number of other widely accepted standards;
2. it has achieved wide acceptance;
3. editors and parsers are available on a wide variety of hardware platforms and operating systems (*cf.* requirement 6) and
4. it was designed to provide structured documents and thus matches the requirements of our test specifications well.

XML documents – laid out with some simple rules – are easy to read and interpret. Understanding is made easier by several readily available editors which highlight the structure and provide various logical views - satisfying part of requirement 7.

By defining a minimal number of elements in the test specification, we kept it simple and easy to use (*cf.* requirement

2); rather than make the test specification complex, we allow testers to write 'helper' classes in the language of the system being tested. This gives testers all the power of a programming language (which they presumably already know!) and avoids the need for them to learn an additional language solely for testing.

The specification uses the terminology of object oriented designs and targets individual methods of a class. However, it can equally well describe test sets for functions written, for example, in C or Ada. As long as there is a well-defined interface, `<MethodCall>` elements can be constructed.

The `Pre` ('Prefix') attribute of an operation allows a tester to specify an operation which creates a common initial environment for multiple tests and invoke it by name as needed in other operations. The `<Invariant>` element allows a tester to specify a method that will be invoked when any object of a class is modified or constructed. Invariants may be specified at any level and the usual scope rules apply: a local invariant for a particular class may be specified to override a global one. Making the test pattern verifier automatically invoke the invariant every time an object of a particular class is modified relieves testers from the tedium of explicitly adding invariant checks at many places and contributes to robust testing (*i.e.* testing that detects errors at the earliest possible point in an operation) by automatically invoking checks at every relevant point that a tester might be tempted to omit in favour of a single check at the end of an operation. The `Pre` and `<Invariant>` capabilities satisfy requirement 5.

A single test specification contains a hierarchy of elements designed to make regression testing after minor maintenance exercises simple and efficient. A test specification itself may contain a number of `<TestSet>` elements which may contain either `<TestGroup>`s or `<Operation>`s. `<TestGroup>`s may contain `<Operation>`s or nested `<TestGroup>`s. An `<Operation>` defines a single test and may consist of a number of `<Constructor>` or `<MethodCall>` elements. (The distinction between constructors and other methods is made for simplicity in specifying tests for object oriented languages such as Java and C++. For other languages, it can be ignored.)

We suggest that a `<TestGroup>` contains `<Operation>`s which target a single method of a class whereas a `<TestSet>` might contain the tests for a single class. This would allow a maintainence programmer, having made changes to one method of a class, to immediately verify that the changes were correct by selecting a `<TestGroup>` targetting the modified method and running all the tests in it. Having obtained passes for all the tests in the `<TestGroup>` most likely to be sensitive to the changes, all the remaining `<TestSet>` s – which may need to be run to meet an organisation's standards – can be run with lower priority in the background or on a machine set aside for long batch runs. However, our association of `<TestSet>` with a class and `<TestGroup>` with a method is just a recommendation and the tester may use the hierarchy in any way appropriate to the system being verified.

Each constructor or method call may have arguments and return a result. Results can be checked against expected values or stored for verification by 'helper' methods. The ability to call 'helper' methods means that the test specification itself can be kept simple and portable: no language-specific features need to be added. This has an additional benefit: implementations of essentially the same function in different languages can use the same test specification, enabling re-use of the significant effort that needs to be invested in test set generation.

We built a 'Test Pattern Verifier' to interpret and process a test specification (*cf.* section III), but since the specification is open and standard, other testers may readily build equivalent tools.

B. Test Results

Results (either return values or altered object state) from method invocations may be used in various ways:

- they may be passed to other methods which check their correctness. In this case, they are assigned a name and marked 'Save' in the `<Result>` element.
- They may be compared against an expected value which is stored in an `<Exp>` element. Discrepancies will be reported as test failures.

Expected values themselves may have different sources:

- They are derived directly from the specification,
- An Automatic Test Pattern Generator (ATPG), such as our symbolic execution system[7], has generated an input test pattern which has been used to execute the method-under-test and produce a result. The result has been checked for compliance with the specification.
- An ATPG-generated input has been used to produce a result which is 'reasonable', *i.e.* it has passed cursory checks (*e.g.* no exception was generated and the value is within an acceptable range) for correctness,
- An ATPG-generated input produces a result which has not been checked.

Values returned by method invocations may be either *specified* or *calculated*. Specified results are derived directly from a component's specification – which may contain an exact value or a method for computing the value – and are stored in `<Exp>` elements of a test specification. Calculated results are determined by execution of the component's code and are stored in separate `<ResultSet>` documents which accompany a test specification.

In both cases, discrepancies between specified or calculated results are flagged as errors or potential errors by the test pattern verifier (*cf.* section III) when the tests are run.

III. Test Pattern Verifier

Component users must be able to run the tests described in a test specification. We have developed a lightweight, portable program – the test pattern verifier (TPV) – which reads XML test specifications, applies the tests to a component and checks results against those in `<Exp>` elements or

previously stored in `<ResultSet>` documents by the TPV. The TPV is written in Java and is small enough (136kb of Java bytecodes + 451kb for a SAX XML parser[1] in compressed (zip) files) that it does not place an undue burden on a file system or communications links (when downloaded as part of a code/documentation/test certificate package). If many component developers adopt a standard test specification, then this overhead only occurs once on a developer's system.

However, neither a developer nor a purchaser are constrained to using our TPV. Parsers for XML are readily available and either party may easily construct a TPV program to meet their own requirements

IV. Discussion

Our approach which involves developers providing their own test data to component purchasers has many advantages over the certification laboratory approach:

1. Costs are reduced: the incremental cost to developers is small. They have produced extensive tests as part of their own verification procedures; without them, they cannot make any claim for reliability.
2. There is no need for trust: purchasers are supplied with the test data, means to interpret it (the XML DTD can be used by most XML editors to display the test specification's structure and content) and a means to run the tests and verify that the developer's claims for correctness are sustainable.
3. Any reliability level claimed by the developer as a result of his testing can be confirmed by examination the tests for conformance with the purchaser's understanding of the specification and completeness.
4. The test specifications augment the – usually natural language and therefore laden with potential for ambiguity – functional specifications. The test specifications and accompanying actual results provide a precise (if voluminous!) specification for the actual behaviour of the component.

The test specifications add considerable value to a software component: in many cases they already exist in collections of test programs, scripts and test procedures (*i.e.* instructions for the execution of manual tests). All that is needed is a standard format in which they can be 'packaged' and supplied with a component to a purchaser.

Voas' examples imply that proposal targets large application software suites. Our proposal, on the other hand, is designed for 'component'[2] level software. We find that complete test specifications are usually several times as large as the components they test. For example, the test specifications for a small component (a Heap) in Java require 15.3kB, whereas the fully commented source code requires 9.2kB, a 1.7:1 ratio. This ratio *increases* as the size of a component increases. (We postulate that it will prove in practice to be at least $O(n^2)$.) Thus the volume of a test specification that approached any definition of 'complete' needed to accompany a large application would be impractical. Although the amount of operational data required by an SCL to issue a certificate will have a similar complexity. Using code instrumented for residual testing[2] will only reduce the constant factor!

In addition, we note that the test specifications provide a valuable input to an SCL preparing to certify a component.

V. Conclusion

In addition to reliability, CBSE will need economic sources of components. Component software presents an opportunity for many small developers to produce high quality software and find a market for it. These small developers can operate very efficiently by operating with low overheads, specializing in certain application domains or otherwise leveraging particular skills or knowledge. Since it is always going to be very difficult to predict whether any one component will become popular with other developers, small developers will be reluctant to incur additional costs speculating that one component will actually achieve some reasonable volume of sales. Third party SCL's will only add to costs unnecessarily – and be impractical for small developers. If CBSE practitioners see that the only way to obtain reliable components is to use ones certified by SCLs, the industry may well stifle itself before it has a chance to develop its full potential.

However, if component authors generate complete (or substantially complete) test sets and supply them with components, they incur little additional cost, since they must generate the tests in the first place. Any extra effort is also adding value to a component – as a tested component is certainly a more marketable commodity – with relatively small investments in additional time. SCL certification would also add value to a component, but it is likely that many more sales would be needed to recover the cost of generating the additional value in this way. The test specifications that we are proposing have similarities with data sheets supplied by manufacturers for integrated circuits. This market is well established and thus a good indicator for successful practices.

As a final note, we believe that SCL's do have a place: there will be complex or valuable components destined for systems for which reliability is an overriding goal, so that third party certification – possibly using author-generated test sets – is economically justified, but we consider that such situations are not likely to be common compared to the majority of commercial software developments where the cost of failure is not large and only a small premium (the cost of generating tests and inspecting them) for reliability can be justified.

VI. Acknowledgments

This work was supported by a grant from Software Engineering Australia (Western Australia) Ltd through the Software Engineering Quality Centres Program of the De-

[1] This figure applies to the SAX parser section of Oracle's parser[8]: other parsers may require even less.

[2] Remember that our use of the term 'component' is extremely broad: encompassing all the definitions collected by Sampat[5].

partment of Communications, Information Technology and the Arts.

Appendix

I. Test Specification DTD

The full document type definition of our test specification is reproduced here: comments have been removed to satisfy the constraints of this publication. An extensively commented version may be found on the Software Component Laboratory web site[9].

```
<?xml version="1.0" encoding="utf-8"?>
<!ELEMENT TestSpecification ( TestSet* ) >
<!ELEMENT TestSet (Desc? (Operation | TestGroup | Invariant )* ) >
   <!ATTLIST TestSet Name ID #REQUIRED >
  <!ELEMENT TestGroup (Desc? (Operation | TestGroup | Invariant)* ) >
    <!ATTLIST TestGroup Name ID #REQUIRED >
    <!ATTLIST TestGroup TargetMethod CDATA #IMPLIED >
  <!ELEMENT Invariant ( Arg* (Result | Exception)? ) >
    <!ATTLIST Invariant DataType CDATA #REQUIRED >
    <!ATTLIST Invariant MethodCall CDATA #REQUIRED >
   <!ELEMENT Operation ( (Constructor | MethodCall | Invariant)* ) >
    <!ATTLIST Operation Name ID #REQUIRED >
    <!ATTLIST Operation Pre IDREF #IMPLIED>
    <!ATTLIST Operation Version CDATA #IMPLIED>
    <!ELEMENT Constructor ( Arg*, (Result | Exception)? ) >
      <!ATTLIST Constructor Name CDATA #REQUIRED >
    <!ELEMENT MethodCall ( Arg*, (Result | Exception)? ) >
      <!ATTLIST MethodCall Name CDATA #REQUIRED >
      <!ATTLIST MethodCall Target CDATA #REQUIRED >
      <!ATTLIST MethodCall Static ( Y | N ) "N" >
      <!ELEMENT Arg (#PCDATA) >
        <!ATTLIST Arg Name CDATA #IMPLIED >
        <!ATTLIST Arg Source CDATA #IMPLIED >
        <!ATTLIST Arg DataType CDATA #IMPLIED>
      <!ELEMENT Result (Exp?) >
        <!ATTLIST Result Name CDATA #IMPLIED>
        <!ATTLIST Result DataType CDATA #IMPLIED>
        <!ATTLIST Result Qualification CDATA #IMPLIED >
        <!ATTLIST Result  Save ( Y | N ) "N" >
        <!ELEMENT Exp (#PCDATA) >
          <!ATTLIST Exp SpecVersion CDATA #IMPLIED >
      <!ELEMENT Exception (Exp?) >
        <!ATTLIST Exception Name CDATA #IMPLIED >
        <!ATTLIST Exception DataType CDATA #REQUIRED >
        <!ATTLIST Exception Qualification CDATA #IMPLIED >
        <!ATTLIST Exception  Save ( Y | N ) "N" >
```

References

[1] Jeffrey Voas, "Developing a usage-based software certification process," *IEEE Computer*, vol. 33, no. 8, pp. 32–37, aug 2000.

[2] Christina Pavlopoulou and Michal Young, "Residual test coverage monitoring," in *Proceedings of the 1999 International Conference on Software Engineering*. 1999, pp. 277–284, IEEE Computer Society Press / ACM Press.

[3] Radio Technical Commission for Aeronautics, *Software Considerations in Airborne Systems and Equipment Certification: DO-178B*, RTCA, Inc, 1992.

[4] Ministry of Defence Directorate of Standardisation, *Defence Standard 00-55: The Procurement of Safety Critical Software in Defence Equipment*, HM Government, 1997.

[5] Nilesh Sampat, *Components and Component-Ware Development; A collection of component definitions*, http://www.cms.dmu.ac.uk/ nmsampat/research/subject/reuse/components/index.html, 1998.

[6] W3 Consortium, *XML, Version 1.0*, w3c, 1998.

[7] Gareth Lee, *Symbolic Executor for the CTB: work in progress*, Software Component Laboratory, CIIPS, University of Western Australia, 2000.

[8] Oracle Technology Network, *Oracle's XML Parser for Java v2*, `http://otn.oracle.com/tech/xml/parser_java2`, 2000.

[9] Centre for Intelligent Information Processing Systems, UWA, *Software Component Laboratory*, `http://ciips.ee.uwa.edu.au/Research/SCL/SCL.html`, 2000.