Software Component Certification

John Morris
Gareth Lee
Kris Parker
Gary A. Bundell
University of Western Australia
Chiou Peng Lam
Murdoch University

Although the stringent requirements of some critical applications may require independent certification, the authors see developer self-certification as a viable alternative in many other cases.

Most current methods for certifying software are process-based and require—according to Jeffrey Voas—that software publishers "take oaths concerning which development standards and processes they will use."[1] Voas, among others, 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's scheme, SCLs would

- accept instrumented software from developers,
- pass the instrumented software along to prequalified users,
- gather information from user sites,
- use data gathered from several sites to generate statistics on use and performance in the field, and
- provide limited warranties for the software based on these statistics.

Thus, the SCLs assume the role of independent auditors that monitor the processes developers follow and provide statistics about how their clients' products perform. Additionally, the SCLs could, by continuing to collect data over time, broaden the warranty as the software's operational profile broadens.

We accept that using SCLs may work well for certain software distribution models, but we also observe that it cannot be applied to all software development. As the "Software Certification Laboratory Limitations" sidebar indicates, this approach has several drawbacks. For example, an SCL may work well for larger software houses that ship mass-marketed software applications to the public, but it is less satisfactory for smaller developers who make reusable components or safety-critical software or for developers who belong to the freeware community.

DEVELOPER SELF-CERTIFICATION

We propose an entirely different model for software component certification, one based on test certificates that developers supply in a standard portable form so that purchasers can, in short order, determine the quality and suitability of purchased software. For our test specifications, the term *component* refers to any piece of software with a well-defined interface. This meaning thus encompasses components that satisfy any of the term's many definitions found in the literature,[2] 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 suc-

0018-9162/01/$10.00 © 2001 IEEE

cessful integrated-circuit market—even referring to components as software ICs. When describing an integrated circuit's capabilities, manufacturers have adopted a fairly standard format: Most data sheets are structurally similar, divided into AC and DC characteristics. The AC sections report propagation delays, the DC sections specify voltage levels and power consumption: All manufacturers use similar notations and structures in both sections. This de facto standardization offers great benefit to design engineers: Having a standard structure for data sheets from different manufacturers makes them generally quick and easy to comprehend.

Our approach lets developers employ software process models, but does not require it. Clearly, adopting process models—such as those developed by the Software Engineering Institute at Carnegie Mellon University—can enhance the resulting products' qual-

Software Certification Laboratory Limitations

Using software certification libraries brings with it several limitations in the areas of cost, liability, developer resources needed to access SCLs, and applicability to safety-critical systems.

Cost

Of necessity, SCLs add significantly to the certified product's overall cost. Part of that cost is for insurance—necessary for the certifiers to provide any form of guarantee to purchasers. The only benefit to a purchaser is that insurers may be prepared, presumably after some sufficiently long interval, to factor the amount of operational data available into their calculation of risk and, therefore, the premium.

Liability

To be effective, third-party certifiers must provide some form of precisely stated warranty: Purchasers cannot be expected to pay a premium for their services without additional value. While testing a component certainly adds value, we doubt whether testing alone—without some form of guarantee of its completeness—will add sufficient value to make an SCL viable. User-based testing essentially amounts to random testing, albeit biased to some operational profile. Such testing exposes an SCL that provides a warranty to significant damage claims from time bombs embedded in code that has never been exercised. SCLs thus may occupy an even more invidious position than developers. Developers simply disclaim liability; SCLs provide professional advice to clients on the risk of using a component. Courts have not been kind when such advice has been proven faulty.

Developer resources

Much successful, widely used software—such as Linux and Free Software Foundation software—is written by single programmers or small programming groups working independently. We would expect their products 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

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

Lacking a viable alternative to SCLs, many developers will forgo using software developed by these small independents.

Safety-critical systems

Voas notes that safety-critical systems present an area of special challenge, recognizing that SCLs will have some difficulty persuading testers to "fly uncertified software-controlled aircraft or use uncertified software-controlled medical devices." His solution is to certify the software in noncritical environments first: He suggests that, once a product achieves noncritical certification, it could be used with confidence in safety-critical applications that mirror the certification environment.

This approach manifests several problems. First, many key components of safety-critical systems will have no application in noncritical systems. Second, the operational profiles collected from users are unlikely to satisfy the software product standards used in this area. For example, they would not be expected to satisfy coverage criteria required for airborne[1] or defense systems.[2] Thus, we still need testing to satisfy the applicable standard. Further, such testing will likely consume the major part of any testing budget, as even ensuring statement coverage requires the construction of test cases for many rare situations unlikely to be covered in any reasonable period of actual use.

Testing alternatives

Our model clearly targets small components and does not eclipse all uses of the SCL model. For example, in the many applications where the cost of software failure is high, third-party certification by specialist testing organizations provides the only acceptable model. Further, SCL certification may be the only practical alternative for large software systems.

References

1. Radio Technical Commission for Aeronautics, *Software Considerations in Airborne Systems and Equipment Certification: DO-178B*, RCTA, Washington, D.C., 1992.
2. British Ministry of Defense Directorate of Standardisation, *Defense Standard 00-55: The Procurement of Safety Critical Software in Defense Equipment*, 1997; http://wheelie.tees.ac.uk/hazop/standards/55/mainpage.htm (current 6 Aug. 2001).

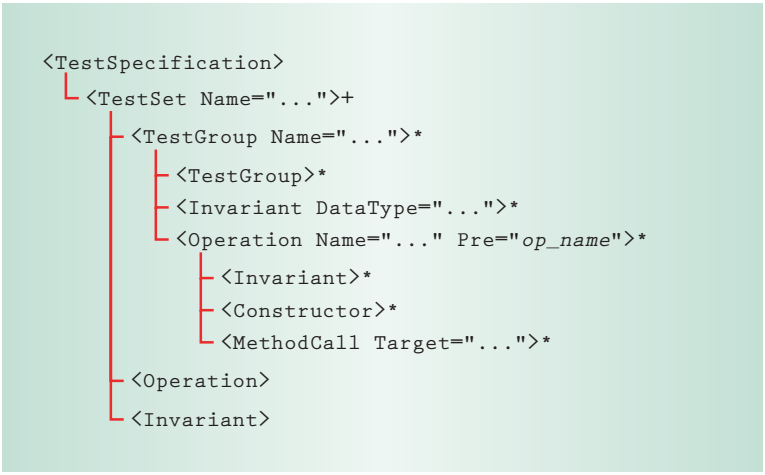

Figure 1. Elements of an XML test-specification grammar, displayed in tree form. A test specification can contain several TestSet elements, which contain either TestGroup or Operation elements. Further, TestGroup elements can contain Operation elements or nested TestGroup elements.

ity. We must balance against that benefit, however, the increased staff training costs and overhead that implementing the process incurs.

STANDARD TEST SPECIFICATIONS

If developers are to supply test sets to purchasers, they will need a standard, portable way of specifying tests so that a component user can assess how much testing the component has undergone. Potential customers can then make an informed judgment about the likely risk of the component failing in their application, keeping in mind the nature of the tests and the intended application.[3]

To fill this role, we designed a test specification that aims to be

- standard and portable;
- simple and easy to learn;
- devoid of language-specific features;
- equally able to work with object-oriented systems, simple functions, and complex components such as distributed objects or Enterprise JavaBeans;
- efficient at handling the repetitive nature of many test sets;
- capable of offering widely available and easily produced test-generation tools that do not require proprietary software;
- free of proprietary-software requirements for interpreting and running the tests; and
- able to support regression testing.

We based our test pattern document format on the W3C's Extensible Markup Language, which satisfies most of our requirements.[4] XML is a widely adopted general-purpose markup language for representing hierarchical data items. We have defined an XML grammar, specialized for representing test specifications, published in the form of a document type definition (DTD) that can be downloaded from our Web site.[5] Figure 1 depicts, in tree form, the elements within our grammar.

XML is well suited to representing test specifications because it adheres to a standard developed by an independent organization responsible for several other widely accepted standards. It has achieved broad acceptance across the industry, leading to the development of editors and parsers for a variety of platforms and operating systems. Further, XML's developers designed the language to provide structured documents, which support our test specifications well.

XML documents—laid out with some simple rules—can be read and interpreted easily. Several readily available editors make understanding the language easier by highlighting its structure and providing various logical views.

To keep the test specification simple and easy to use, we defined a minimal number of elements for it. Rather than adding elements to support high-level requirements, we allow testers to write *helper* classes in the language of the system they are testing. This approach gives testers all the power of a programming language they presumably already know and avoids forcing them to learn an additional language solely for testing.

SPECIFICATION DOCUMENT FORMAT

The specification uses the terminology of object-oriented designs and targets a class's individual methods. However, it can describe test sets for functions written using non-OO languages such as C or Ada equally well. As long as a well-defined interface exists, a tester can construct MethodCall elements.

An operation's prefix attribute lets a tester specify an operation that creates a common initial environment for multiple tests, then invoke it by name as needed in other operations. The Invariant element lets a tester specify a method that the system will invoke when he modifies or constructs any object of a class. The tester can specify invariants at any level, and the usual scope rules apply: A local invariant for a particular class may be specified to override a global one. The test pattern verifier automatically invokes the invariants every time the tests are run after a developer changes a class's methods, thereby sparing testers the tedium of explicitly adding invariant checks at many points. This invocation contributes to robust testing—which calls for detecting errors at the earliest possible point in an operation—by automatically invoking checks at every relevant point where a tester might be tempted to omit them in favor of a single check at the end of an operation.

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 can contain several TestSet elements, which can contain either TestGroup or Operation elements. Further, TestGroup elements can contain Operation elements or nested TestGroup elements. An Operation defines a single test and can consist of several Constructor or MethodCall elements.

We make the distinction between constructors and other methods to simplify specifying tests for OO languages such as Java and C++. For other languages, we can ignore the distinction. Figure 2 shows the elements of our grammar associated with method invocation.

We suggest that a TestGroup should contain Operation elements that target a single method of a class, whereas a TestSet could contain the tests for a single class. This would allow a maintenance programmer, having made changes to one method of a class, to immediately verify that the changes were correct by selecting a TestGroup targeting 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 elements 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 a recommendation only. The tester can use the hierarchy in any way appropriate to the system being verified.

Each constructor or method call can have arguments and return a result. The test harness can check the results against expected values or store them for verification by helper methods. The ability to call helper methods means that the test specification itself can be kept simple and portable: It isn't necessary to add any language-specific features. An additional benefit is that implementations of the same function in different languages can use the same test specification, which reuses the significant effort invested in test set generation.

The "Sample Test Document" sidebar contains a sample test specification that can be applied to the Vector class, which Sun Microsystems supplies as part of the Java API. We have built a test-pattern verifier application to interpret and process a test specification, but because the specification is open and standard, other testers can readily build equivalent tools.

TEST RESULTS

The specification can use the results from method invocations—either return values or an altered object state—in various ways. It can pass the results to other methods that check their correctness, in which case the specification assigns them a name in the Result element; or it can compare the results against an expected value stored in an Exp element, in which case discrepancies will be reported as test failures.

Expected values themselves can have different sources. They can be derived from

- the specification;
- an automatic test pattern generator (ATPG), such as our symbolic execution system,[6] which generates an input test pattern to execute the method under test, producing a result that the ATPG has checked for compliance with the specification;
- an ATPG-generated input used to produce a result that passed cursory checks for correctness, meaning that it generated no exceptions, and a value within a plausible range; or
- an ATPG-generated input that produced a result that has not been checked.

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

Figure 2. Grammar elements associated with method invocation. Each constructor or method call can have arguments and return a result. The test harness can check results against expected values or store them for verification by helper methods.

Method invocations can return either specified or calculated values. Specified results derive directly from a component's specification—which can contain an exact value or a method for computing the value—and are stored in the test specification's Exp elements. Executing the component's code determines the calculated results and stores them in the separate ResultSet documents that accompany a test specification. In both cases, when the testers run the tests, the test-pattern verifier flags discrepancies between specified or calculated results as errors or potential errors.

TEST-PATTERN VERIFIER

Component users must be able to run the tests that a test specification describes. We have developed a lightweight, portable program—the test-pattern verifier (TPV)—that reads XML test specifications, applies the tests to a component, and checks results against those in Exp elements or that the TPV previously stored in ResultSet documents. Written in Java, the TPV is small enough to avoid placing an undue burden on a system when downloaded as part of a code, documentation, and test-certificate package. The TPV amounts to 136 Kbytes of Java bytecode, but a SAX (Simple API for XML) parser requires an additional ~451 Kbytes, both delivered as compressed files. The quoted size applies to Oracle's SAX parser,[7] but smaller parsers are now available. If many component developers adopt a standard test specification, this overhead only occurs once on a developer's system.

Using our TPV limits neither developer nor purchaser. XML parsers are readily available, and either party can easily construct a TPV program to meet its own requirements.

IMPLICATIONS

Our approach, which requires that developers provide their own test data to component purchasers, has many advantages over the certification laboratory approach.

- *Reduced costs*. The incremental cost to developers is small because they have produced extensive tests as part of their own verification procedures. Without such tests, they cannot make any claim for component reliability.
- *Guaranteed trust*. Purchasers receive the test data and the means to interpret it: Most XML editors can use the XML DTD to display the test specification's structure and content. Further, purchasers receive a means for running the tests and verifying that the developer's claims for correctness are sustainable.
- *Confirmed conformance*. To confirm developer claims regarding a given product's testing level, purchasers can review the tests to judge how well they conform to their understanding of the specification.
- *Augmented functional requirements*. The test specifications augment the functional requirements, which are usually natural language and therefore laden with potential ambiguities. The specifications and accompanying actual results provide a precise if voluminous specification of actual component behavior.
- *Added value*. 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—requiring only a standard format for packaging and supplying them with a component to a purchaser.

Voas's examples imply that his proposal targets large application software suites. We designed our proposal, on the other hand, for component-level software,

Sample Test Document

As a simple example of our test specification, we created a heavily abridged test of the java.util.Vector class within the standard Java API. The DocHeader, Implementation, and Interface elements provide information about the attribution of the test set, the specific implementation being tested, and the interface to which that implementation adheres and through which the test should be conducted. The body of the XML test specification follows:

```
<?xml version="1.0"?>
<Component Name="Vector">
   <DocHeader Name="vector.xml">
      <Author Name="Kris Parker" Org="CIIPS">KP</Author>
      <Copyright>2001, CIIPS</Copyright>
      <Created Date="04-07-2001" Who="KP" Ver="1.0"/>
   </DocHeader>
   <Implementation Name="Java" Environ="any_java" Lang="java"
      IntName="java.util.Vector"/>
   <Interface Name="java.util.Vector" TestSetName="add_tests"/>

   <TestSet Name="add_tests">
      <TestGroup Name="Simple tests">
         <Operation Name="New">
            <Constructor Name="java.util.Vector">
               <Result Name="vect" DataType="java.util.Vector"/>
            </Constructor>
            <MethodCall Name="size" Target="vect">
               <Result Name="size" DataType="int">
                  <Exp>0</Exp>
               </Result>
            </MethodCall>
         </Operation>
         <Operation Name="Add" Pre="New">
            <Constructor Name="java.lang.Object">
               <Result Name="obj" DataType="java.lang.Object"/>
         </Constructor>
         <MethodCall Name="add" Target="vect">
               <Arg Source="obj" DataType="java.lang.Object"/>
```

which we define as encompassing all the definitions Nilesh Sampat collected.[2] 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 in Java, a Heap, require 15.3 Kbytes, whereas the fully commented source code requires 9.2 Kbytes, a 1.7 to 1 ratio. This ratio increases as a component's size increases. Thus, the volume of the test specification necessary to accompany a large application, and that approaches any definition of complete, would be impractically large. The amount of operational data an SCL requires to issue a certificate would have a similar complexity. Using code instrumented for residual testing[8] only helps reduce the constant factor. Further, the test specifications also provide a valuable input to an SCL preparing to certify a component, so either approach requires them.

In addition to reliability, component-based software engineering (CBSE) will need economical component sources. Component software presents an opportunity for many small developers to produce and market high-quality software. These developers can compete efficiently by operating with low overheads, specializing in certain application domains, or otherwise leveraging particular skills or knowledge.

Because predicting whether any one component will become popular with developers will always be difficult, small developers will be reluctant to incur additional costs by speculating that one component will actually achieve some reasonable sales volume. Third-party SCLs will only add to costs unnecessarily, and they are impractical for small developers. If CBSE practitioners see that they can obtain reliable components only by using those that SCLs certify, the indus-

```
        </MethodCall>
        <MethodCall Name="size" Target="vect">
          <Result Name="size" DataType="int">
            <Exp>1</Exp>
          </Result>
        </MethodCall>
        <MethodCall Name="get" Target="vect">
          <Arg DataType="int">0</Arg>
          <Result Name="x" DataType="java.lang.Object"/>
        </MethodCall>
        <MethodCall Name="equals" Target="obj">
          <Arg Source="x" DataType="java.lang.Object"/>
          <Result Name="equal" DataType="boolean">
            <Exp>true</Exp>
          </Result>
        </MethodCall>
      </Operation>
    </TestGroup>
  </TestSet>
</Component>
```

The body of the document defines a single TestSet, named add_tests, which contains a single TestGroup element, Simple tests. The group then defines an operation entitled New that constructs a new instance of a Vector object and tests that its initial size is 0. The subsequent operation uses the New operation as a prefix to create an initial test environment, then executes a sequence of tests that

- creates a Java object, storing it as the environment variable obj;
- invokes the add method on the Vector to add the object to the data structure;
- invokes the Vector's size method to ensure that this value is now 1; and
- invokes the get method to ensure the 0th element can be retrieved from the Vector.

Typically, testers will use a testing tool to generate such documents automatically, but this example demonstrates that our test specifications are human-readable so that, if necessary, testers can use a simple text editor to read or modify it.

try could stifle itself before developing its full potential.

However, if component authors generate complete or substantially complete test sets and supply them with components, they incur little additional cost because they must generate the tests in the first place. Any extra effort also adds value to a component—as a tested component is certainly a more marketable commodity—while demanding a relatively small investment of additional time. SCL certification would also add value to a component, but it is likely that recovering the cost of generating the additional value in this way would require many more sales. The test specifications we propose resemble the data sheets that integrated-circuit manufacturers supply—a well-established market and thus a good indicator of successful practices.

We believe that SCLs *do* have a place: Complex or valuable components destined for systems that require reliability will economically justify third-party certification. Yet we believe that because there will always be commercial software developments for which failure represents a moderate cost, only a small premium for reliability—the cost of generating tests and inspecting them—can be justified in such cases. ✷

Acknowledgments

This work was supported by a grant from Software Engineering Australia (Western Australia) through the Software Engineering Quality Centres Program of the Department of Communications, Information Technology, and the Arts of the Commonwealth Government of Australia.

References

1. J. Voas, "Developing a Usage-Based Software Certification Process," *Computer*, Aug. 2000, pp. 32-37.
2. N. Sampat, "Components and Component-Ware Development: A Collection of Component Definitions," 1998; http://www.cse.dmu.ac.uk/~nmsampat/research/subject/reuse/components/main_index.html (current Aug. 2001).
3. P. Frankl et al., "Evaluating Testing Methods by Delivered Reliability," *IEEE Trans. Software Eng.*, Aug. 1998, pp. 586-602.
4. World Wide Web Consortium, *Extensible Markup Language 1.0*, 1998; http://www.w3.org/TR/2000/REC-xml- 20001006 (current Aug. 2001).
5. Centre for Intelligent Information Processing Systems, Software Component Laboratory, 2001; http://ciips.ee.uwa.edu.au/Research/SCL/SCL.html (current Aug. 2001) and our test pattern DTD http://ciips.ee.uwa.edu.au/Research/SCL/Docs/Component.dtd (current Aug. 2001).
6. G. Lee et al., "Using Symbolic Execution to Guide Test Generation," *Software Component Laboratory Technical Report*, University of Western Australia, 2001; http://ciips.ee.uwa.edu.au/Research/SCL/Docs/SymbolicExec.dvi (current Aug. 2001).
7. Oracle Technology Network, *Oracle's XML Parser for Java, Version 2*, 2000; http://technet.oracle.com/tech/xml/ (current Aug. 2001).
8. C. Pavlopoulou and M. Young, "Residual Test Coverage Monitoring," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 277-284.

John Morris *is a senior lecturer in the Centre for Intelligent Information Processing Systems, University of Western Australia, and director of the Centre's Software Component Laboratory. His research interests include computer architectures and software verification. He received a PhD in optical spectroscopy from the Australian National University. Contact him at morris@ee.uwa.edu.au.*

Gareth Lee *is a senior research fellow in the Department of Electrical and Electronic Engineering, University of Western Australia. His research interests include speech and pattern recognition, artificial neural networks, and software engineering. He received a PhD in electronic engineering from the University of East Anglia. Contact him at gareth@ee.uwa.edu.au.*

Kris Parker *is a research associate with the Australian Research Centre for Medical Engineering, University of Western Australia. His research interests include client-server architectures. He received a BE in electronic engineering from the University of Western Australia. Contact him at kaypy@ee.uwa.edu.au.*

Gary A. Bundell *is a senior lecturer in the Department of Electrical and Electronic Engineering, University of Western Australia. His research interests include software engineering, in particular, real-time distributed information systems design and analysis. Bundell received a PhD in control engineering from Cambridge University. Contact him at bundell@ee. uwa.edu.au.*

Chiou Peng Lam *is a senior lecturer in software engineering in the School of Engineering at Murdoch University. Her research interests include component-based software engineering, functional reasoning and representation, and intelligent process operation management. She received a PhD in active vision from Curtin University of Technology. Contact her at peng@eng.murdoch.edu.au.*