A Software Component Verification Tool

Gary A Bundell, Gareth Lee, John Morris, Kris Parker
Centre for Intelligent Information Processing Systems
Department of Electrical and Electronic Engineering
The University of Western Australia, Nedlands WA 6907, Australia
[bundell,gareth,morris,kaypy]@ee.uwa.edu.au

Peng Lam
School of Engineering
Murdoch University, Murdoch WA 6150, Australia
peng@eng.murdoch.edu.au

Abstract

Component-Based Software Engineering depends on reliable, robust components, since it may omit a unit test phase wholly or partially from the development cycle. This paper describes a tool that allows a component developer to design and run verification tests. In developing components for our library, we found it necessary to provide multiple mechanisms for identifying and capturing tests to overcome the limitations of any single mechanism. Once specified, test specifications and test results are stored in XML documents, providing a standard, portable form of storing, retrieving and updating test histories. One module of our component test bench, the test pattern verifier, has been designed to be general, lightweight and portable, so that it can be packaged with a component and its test specifications. This allows a component user to verify a component's compliance with specifications in a target environment.

1. Introduction

Component-Based Software Engineering (CBSE) is an emerging methodology for software development that aims to compose applications with plug and play software components (custom-built or Commercial Off-the-Shelf (COTS)) in a framework. This paradigm is becoming increasingly important owing to the maturity of several underlying technologies that support building components and developing applications from sets of these components. Changes in the business and organisational context in which these applications are being developed[4] are also driving CBSE forward. Three component infrastructure technologies - OMG's CORBA[19], Microsoft's Component Object Model (COM) and Distributed COM (DCOM)[3] and Sun's JavaBeans and Enterprise JavaBeans[22] - have matured and have become somewhat standardised. These technologies provide the communication and coordination that are required to construct applications from components.
Recent developments in the business and organisational context that are promoting CBSE include[4]:

- the shift in style and architecture of applications away from centralised mainframe-based applications to distributed applications remotely accessible from a variety of client machines,
- the need for business to maintain some stability in the technology supporting its core business and its internal structure in the face of rapid development in these technology areas *and*
- organisations have already invested significant resources in their existing applications and would prefer to reuse their existing investment in developing new applications quickly and reliably.

CBSE differs from conventional reuse of components in a number of ways. First, components are required to have plug and play capabilities. Second, components hide their implementation details and thus their interfaces should be separated from their implementations. Lastly, in order to promote interoperability, components are usually designed on a pre-defined architecture.
The CBSE process has two phases: component development and component integration. The component-based enterprise software process model[1] for application development consists of the following sequential stages:

- Analysis and Component Acquisition,

- Component-Oriented Design,
- Component Composition,
- Integration Test,
- System Test.

Note the absence of a unit testing phase in this methodology. A corner-stone of the CBSE approach is the benefit of using pre-existing, assumed reliable components. This permits faster construction of complex systems with better, *i.e.* more reliable, outcomes. Also, developers often do not have access to the source code and can only work with components as black boxes and are thus limited to identifying faults. On the other hand, the development of components themselves follows a more conventional approach which includes unit testing - the objective of the tool described in this paper.

Integration is an error-prone process, it is necessary to consider the unavailability of complete and correct behavioural specifications for the components, the high level of volatility of components and the mismatch between components for various reasons[12]. The specifications provided for COTS components may not always be complete or correct. In addition, commercial components are often upgraded and this may lead to cases where the upgrades do not have the required capability, retain old bugs and introduce new ones. A major contribution of our work to this process is the coupling of components and test certificates - enabling rapid verification that components do have the required capabilities and that upgrades have not affected correct functions. We define a test descriptor in XML - a simple, portable document standard - which permits test specifications to be read and interpreted on a wide variety of platforms. The tool we describe here contains a test pattern verifier - a light-weight, general and portable module which will run as a stand-alone package. Thus it can be packaged with a component and its test specification - enabling a component user to confirm correct functioning of the component in any target environment.

Many organisations are considering developing large industrial systems that have very high reliability and availability requirements using components [16, 17]. In one reported case, to ensure that the signalling systems that they were developing for the rail transport industry will operate in a fail-safe manner, Profdeta *et al.*[17] used formal methods in their design. This approach can deal with the issues mentioned above to ensure a reliable system but it is not a silver bullet. Despite some reported successes, formal methods are not widely used in industry for a number of reasons[13]. Among these are the needs for standards as well as tools. Wileden[25] suggested that a significant investment in formal methods tools is required for industrial applications just as the significant investment in compiler technology resulted in the widespread take-up of high level languages by the industry. Another approach involves testing the components for each new environment so that developers and users can be confident of the expected behaviour and performance[24]. This approach is not generally feasible as it may incur significant cost.

One important alternative to a method based completely on formal methods is use of a visual modelling language (such as UML) to firstly capture the component requirements, and then to design the component classes and interfaces to meet those requirements[2]. This provides two advantages over the development of components as isolated packages of code to then be verified by a test procedure developed after coding. Firstly, the requirements specification (defined as use cases) can drive the development of test cases and secondly, the provision of a requirements and design model with a component provides the developer using it with a more complete representation of the components capabilities and limitations. It also allows for easier integration into larger design models using the component.

CBSE will not be effectively used until it can be employed within the context of well-understood methods for designing, assembling and maintaining component-based systems[12]. Many of the issues raised here are the subject of current research. A common theme seen in various component-based workshops[14] and research projects[23] is the need for component-based development tools and techniques that can help developers to evaluate and experiment with components.

The component test bench (**CTB**) described in this paper addresses testing tools; it provides a means for developers to generate tests in the first place, for users to verify that components function correctly in some target environment and for both developers and users to run regression tests when components are updated. It provides a mixture of techniques - manual, computer-assisted and automatic - for the generation of tests. This allows techniques such as symbolic execution (*see also* section 5.1), which is very powerful for some cases, but difficult to apply in others, to be used when appropriate. The **CTB** avoids the need to write code for test harnesses: it provides a generic test harness in the test pattern verifier module: developers specify tests which are stored in standard XML documents and run by the test pattern verifier. By using XML for the test specification and Java for the test pattern verifier, we ensure a high degree of portability - compared to systems based on special-purpose scripting languages, such as Buwalda's Action Word method[6] - and provide a lightweight verifier that can be packaged with small components. By integrating test generation (manual, semi-automatic and automated) with test execution, the **CTB** enables them to both use the specification to create basic test cases and allows analysis of written code to identify further tests.

1.1. Outline

In section 2, we define the types of components that **CTB** will handle. Section 3 defines a key concept - the test operation or sequence of calls that make up a single test. This section also describes the test specification that we developed and its use in initial and regression testing (section 6). Section 4 discusses the instrumented run-time system - a core software component of **CTB**. Section 5.1 describes the symbolic execution sub-system and how its output guides the selection of necessary tests. The key area of management of regression testing is described in section 6. Finally, section 7 discusses our experience with the **CTB** to date.

2. Scope

Our definition of a component is intentionally broad; it covers the spectrum from ”pure” dataflow functions to components with all the capabilities associated with the term[5]. As long as there is a clearly defined interface to a component, a test specification may be assembled with the **CTB**. Further, the **CTB** handles situations where a component may have multiple implementations, *e.g.* one in Java and one in C, that perform functions that are logically identical. It also allows an implementation to supply multiple interfaces to a user: this might happen when some specification detail changes but backward compatibility is needed so that existing software does not need to be changed. Different interfaces might also provide subsets or views of a component’s full capabilities that are appropriate to a particular application domain.

3. Test Operations

3.1. Definition

The **CTB** requires a test specification to be prepared which describes implementations of a component, interfaces provided by each implementation and test sets that are appropriate to an interface. Test specifications are stored in XML files - portable, structured documents that may be easily read and processed by other systems.
The key element of a test set is an test operation - a sequence of steps necessary to carry out an invidual test.
Test operations will normally ”target” a method in a component. However, in general, it isn’t possible to test a method in isolation, so a test operation is a sequence of method calls. In contrast to Doong and Frankl[9], who define sequences of messages sent to pairs of objects to bring them into equivalent states, we do not restrict calls to methods of the component under test. A full definition of the test descriptor is available[20].

```
<?xml version="1.0"?>
<!DOCTYPE Component SYSTEM "component.dtd" >
<Component Name="RedBlackTree">
  <DocHeader Name="rbtree.xml">
    <Author Name="John Morris" Org="CIIPS"/>
    <Copyright>2000, CIIPS</Copyright>
    <Created Date="15-05-2000" Ver="1.0"/>
  </DocHeader>

  <Implementation Name="Java" Environ="any_java"
                  Lang="java">
    <Source>RedBlackTree.java</Source>
    <IntName>RedBlackTree</IntName>
  </Implementation>
  <Implementation Name="C" Environ="any_C"
                  Lang="ANSI C">
    <Source>RedBlackTree.c</Source>
    <IntName>RedBlackTree</IntName>
  </Implementation>

  <Interface Name="RedBlackTree">
    <TestSetName>general_test</TestSetName>
    <TestSetName>add_tests</TestSetName>
    <TestSetName>delete_tests</TestSetName>
  </Interface>

  <TestSet Name="general test">... </TestSet>
  <TestSet Name="add_tests">... </TestSet>
  <TestSet Name="delete_tests">... </TestSet>
  <ResultSet>
    ...
  </ResultSet>
</Component>
```

Figure 1. Overall structure of a test descriptor. Ellipses in this diagram indicate sections which have been omitted here for clarity. See figure 2 for the detail of the `<TestSet>` element. Similarly details of the `<ResultSet>` element are shown in figure 7.

A typical example is shown in figures 1-4. By using the XML standard for the test specification, we aim to provide a standard interchange format for test specifications: hitherto, test specifications have usually taken the form of scripts that are specific to a single tool. Developers using CBSE approaches need to be able to verify the reliability of a component in a target environment, so a standard interchange format for test specifications is needed. XML provides that - our tool can use any third party SAX[15] parser to parse the test specification; users can use generally available XML editors and processors to read, interpret and modify the test data sets. In this example, the component being tested is a red-black tree[7]. It has a constructor which accepts a list of items which are to be stored in the tree. To construct this list, we use a ‘helper’ class - a simple utility class designed to assist testing - which holds a list of integers, `IntList`. One of `IntList`’s constructors accepts a string and parses it to extract a list of integers. Thus the test operation starts with an invocation of the `IntList` constructor which is passed the string in the `<arg>` element of the test descriptor (*cf.* figure 2). The list constructed is

then passed to the `RedBlackTree` constructor. For the red-black tree, checking the internal structure of the object produced is not practical: it would involve constructing a tree manually - a time consuming and error-prone process. It is much easier to add a method to the class, `Integrity`, which checks the rules for a red-black tree (see Cormen *et al* [7]) and returns true if the tree has all the required properties. This approach depends on the correctness of the `Integrity` method and thus might be seen as a weak one: but the `Integrity` method is relatively simple to write and the approach is really checking the consistency of the `Integrity` method and other code - a tester will not stop until *all* methods are returning the expected results. As a further check that items are added to the tree, we also call the `FindKey` method to verify that the value "8" is able to be found: `FindKey` should return true in this case (*cf.* figures 2-4).

This example shows that, in general, other methods will be needed in a test operation to

- construct objects which are needed to test the target method and
- verify the correct operation of the target method.

As reported by Binder and others[2], we also found that we would often use a sequence of method calls such as those in figure 2 as a *predecessor* to other tests, *e.g.* in the `RedBlackTree`, that sequence could be used to create a structure to test deletion methods. To avoid multiple repetitions of such sequences, a test operation may contain a `<Pre>` element, *e.g.*

```
<Pre Name="add in order"/>
```

which specifies another test operation which should be run as a predecessor to this one. This enables common predecessor test operations to be specified once and used many times in subsequent testing.

Test operations are labelled with the version of the specification for which they were generated. This provides cross-references between test descriptors and design documentation. Expected results which were generated are linked to an actual implementation and are marked with the implementation's version number.

Using XML for the test descriptor files provides portability and allows component users to gain more information about the component's specification by either reading the test descriptor files directly or by using suitable XML tools (which understand the structure of the document) to walk through the tests. The test descriptor files are updated with test results as the **CTB** runs (*cf.* figure 7) so they may be used in regression testing also - providing a means of highlighting any differences in function between an original and updated version of a component. The structured nature of XML documents makes them more suitable for capturing test procedures *and* test results than the test scripts (with a unique syntax) employed by other tools[11].

```
<TestSet Name="general test">
 <TestGroup Name="add tests">
  <Operation Name="add in order">
    <Constructor Name="IntList">
     <Arg DataType="String">2 5 8 12</Arg>
     <Result Name="test_list"></Result>
    </Constructor>
    <Constructor Name="RedBlackTree">
     <Arg DataType="IntList"
          source="test_list">
     </Arg>
     <Result Name=rbtree></Result>
    </Constructor>
    <MethodCall Name="Integrity"
          Target="rbtree">
     <Result Name="valid" DataType=boolean
           Save="y">
      <Exp>true</Exp>
     </Result>
    </MethodCall>
    <MethodCall Name="FindKey"
          Target="rbtree">
     <Arg DataType="String">8</Arg>
     <Result Name="found" DataType="boolean"
           Save="y">
      <Exp>true</Exp>
     </Result>
    </MethodCall>
  </Operation>
 </TestGroup>
</TestSet>
```

Figure 2. Detail of a `<TestSet>` element from an XML test descriptor file. See figure 1 for the overall document structure.

3.2. Hierarchies of Tests

For convenience in management, test operations in test sets may be organised into test groups. This is particularly useful when running regression tests. A programmer who changes, say the addition function in a lookup table or database, may quickly choose a test group consisting of test operations which exercise an add method for a test run. This will rapidly either provide a strong indication that the changes have not affected correctness or identify a problem. If the test groups related to the change do not identify problems, the programmer is able to make a reasonable presumption that the changes are correct and run the full test set needed for the changed component with lower priority - in the background or at a time more convenient for long processing runs such as at night. This is an important consideration for large components where full verification runs may involve thousands or millions of test operations - most of which are unlikely to be affected by the change, but where the complete set must nevertheless be run to ensure no unexpected side effects have been introduced by the

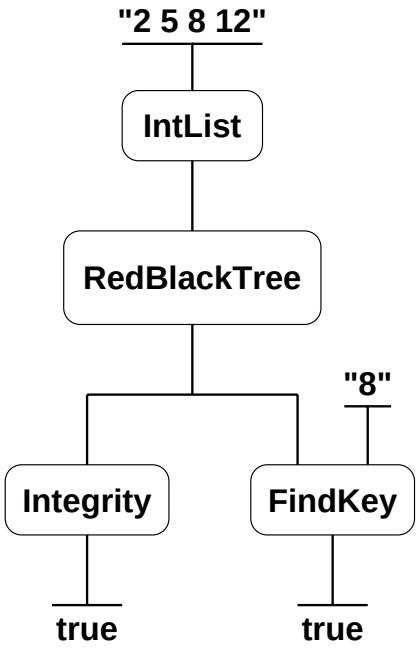

Figure 3. Dataflow representation of a test operation. Data flow is top-to-bottom: constants above the bars are inputs, constants below them are expected results which are checked and stored. Note that this example makes use of the normal dataflow firing rule in that methods `Integrity` and `FindKey` can fire as soon as their data is available and thus in any order.

change.

3.3. Definition of Test Operations

Tests vary in their structure, style and complexity: thus one input ”style” will not suit all. So we allow test operations to be defined in a number of ways:

1. the XML descriptors can written directly with a text editor,
2. an XML editor can use a schema to guide data entry,
3. a conventional GUI with input boxes for each element,
4. a dataflow visual editor (see figure 3) and
5. a program fragment(using a subset of Java, see figure 4) is written using a text editor.

In addition, the symbolic execution system (see section 5.1) can generate or suggest test cases.

```
IntList t_list =
        new IntList( "2 5 8 12" );
RedBlackTree rbtree =
        new RedBlackTree( t_list );
boolean valid = rbtree.Integrity();
expect( valid == true );
boolean found = rbtree.FindKey("8");
expect( found == true );
```

Figure 4. Text representation of the test in figure 3. The syntax is a subset of Java: the `expect` function is rather like a C `assert`, but it also triggers the checking and storage of the result in the `<ResultSet>` element for regression testing.

Test operations define the context for a test by creating a set of objects which constitute the environment in which the target method runs. For example, in figure 4, an environment is created which contains variables with names: `t_list, rbtree, valid` and `found`. In the dataflow diagram in figure 3, the system assigns names to each arc.

3.4. Test Operation Results

Results expected from test operations may be derived in various ways:

- the result is part of the specification or directly derivable from it by independent calculation,
- the result is computed using either the **IRTS**, a Java virtual machine or by running a C or C++ program.

Test inputs are also derived in various ways:

- a part of the specification calls for a specific test,
- the developer performs an informal or formal equivalence class analysis on either the specification or the code or
- symbolic execution suggests a set of inputs, *e.g.* in figure 6 a value of `arg1` in (3,5] is being suggested.

This leads to a classification of expected results from method calls within test operations:

1. **Specified** (strongest, user input, considered part of the specification, equivalent to a UML use case),
2. **Strong accept** (input from symbolic execution, result computed, user has marked it correct, *i.e.* has independently verified that the result conforms to the specification),

3. **Weak accept** (input from symbolic execution, result computed, user has marked it acceptable, but has presumably skipped a perhaps long and detailed calculation to verify conformation with the specification),

4. **Pending** (input from symbolic execution, result computed but not checked, assumed correct),

5. **Intermediate** (needed for a subsequent method call but not checked) and

6. **Unknown** (input from symbolic execution, result not computed yet)

Note that our system allows incremental development, *e.g.* it allows testers to use the symbolic executor to generate test operation data and store it without running test operations (which may take a considerable time and therefore ideally be deferred). ”Strong accept” and ”specified” are essentially equivalent, but we retain the distinction to identify the original source of the test.

4. Instrumented Run-Time System (IRTS)

Once a test operation has been defined, it can be ”run” in two ways:

1. on the **CTB**’s internal Instrumented Run-Time System (**IRTS**)

2. on one of the ’standard’ virtual machines (for Java) or by compiling and executing (C, C++)

The **IRTS** is a Java execution environment[1] designed specifically for testing software components. It has the following capabilities:

- It is able to precisely profile execution of methods. Conventional profilers operate by periodically sampling the program counter. This only gives a statistical sample of the statements executed rather than a precise set. When testing systems are used to create statement coverage metrics they need to know with absolute certainty whether a specific statement was visited (or path was followed) for a particular test case.

- It is able to support different execution models. Traditional VMs provide literal execution in that they allow actual values to propagate through their dataflow paths. To analyse the operation of certain algorithms it is often more appropriate to execute symbolically allowing algebraic values to propagate. These capabilities are described in the following section.

- It is able to test additional execution conditions. Without including extra instructions within the execution sequence, it is impossible with a conventional VM to monitor certain aspects of program behaviour. For instance, many automatic test pattern generation mechanisms need to be able to obtain the values and functional form of the conditions that control the flow of execution[10]. The instrumentation capability of the **IRTS** allows it to collect information about the implementation under test during execution.

- It is able to track data flow during execution. Much dataflow analysis can be performed statically. However the polymorphism features in object oriented languages sometimes make it impossible to reliably ascertain dataflow across method boundaries: event handlers commonly cause this problem. Use of anonymous inner classes in Java applications also means it is often only possible to determine dataflow during execution.

- It is able to provide tester feedback during execution. The system needs to be able to operate rather like a symbolic debugger by providing single step capabilities, allowing the user to monitor the value of specific variables during operation. This provides direct feedback to the tester showing how failing tests execute. This is particularly useful for testing boundary conditions.

In light of these requirements it was decided to implement the **IRTS** as a Java source interpreter. The approach followed was to create a source parser/interpreter. This clearly affects performance, since it bypasses the compiled byte-codes altogether. It also assumes that the compiler that will be used before the application can run is equivalent to our parser and will faithfully translate the source into byte codes. However, basing the **IRTS** on the byte codes was rejected because the compilation process in Java frequently acts to obfuscate the original structure of the program, whereas source execution allows the **IRTS** to directly analyse the structure of the program using its original lexical expression.

Since the interpreter system is itself implemented in Java, it has the option of calling through to the underlying VM at any point to efficiently run methods that belong to trusted classes, *e.g.* the Java standard APIs would be ‘trusted’. Performance is only lost on those methods of the component which are currently under test. Running the interpreter within a Java VM also ensures that its behaviour is authentic since many interpreted operations can be passed through to the underlying VM to ensure that they behave correctly.

[1] Currently we have only built an IRTS for Java; other languages are planned.

5. Symbolic Execution

Choice of the necessary test cases is a difficult and time-consuming task: in even moderately complex components the number of cases will be quite daunting.
Symbolic execution has a useful role to play in automatic test pattern generation, when combined with a path generation algorithm such as one based on definition-use criteria[18]. It has received little attention because of difficulties in the symbolic execution of code in a number of widely used languages. Java's strict regulation of the way pointers are used alleviates some of these difficulties and suggests that symbolic execution should be revisited. Symbolic execution allows the direct creation of equivalence classes of test patterns by analysis of the conditional structure of the code. For each path followed through a particular method, the symbolic executor will produce an algebraic description of the equivalence class. This can then be used to allow the creation of one or more exemplars for inclusion within the test set.

5.1. Symbolic Execution

Currently, symbolic execution works on single methods in Java classes: if the target method is not the first one in a test operation, then either the **IRTS** or the VM is used to generate the environment in which the target method will run. The user then specifies a path by indicating which branch (true or false) of every choice is taken. Loop terminations are considered single branches, so if a loop is to be executed three times, then the sequence "TTTF" (take the branch back to the start of the loop 3 times, then exit the loop) is entered. The example in figure 6 shows a simple method with a single loop: the tester has entered "TTTF" in the "Exec. Path" box to direct the symbolic executor to iterate through the loop 3 times.
The output from symbolic execution is an expression defining an equivalence class on the inputs to the method. The user can then either select a set of values for the inputs which satisifies the expression or allow the system to choose one.
Initially the test operation involving this set of inputs would be marked "pending". If the user enters an expected result, then this set will be marked "specified": it has the same status as one input initially. However, if the method is run first and an output produced, then - unless the tester rejects the result entirely (because running the method with the chosen inputs has produced a result which is clearly wrong) - the result is initially marked "weak accept". The tester may upgrade this result to strong accept after having confirmed that the result does, in fact, match the specification.
At the time of writing, testers must explicitly enter paths: it is planned to add automatic generation of paths using criteria such as defintion-use paths[18] or coverage heuristics.

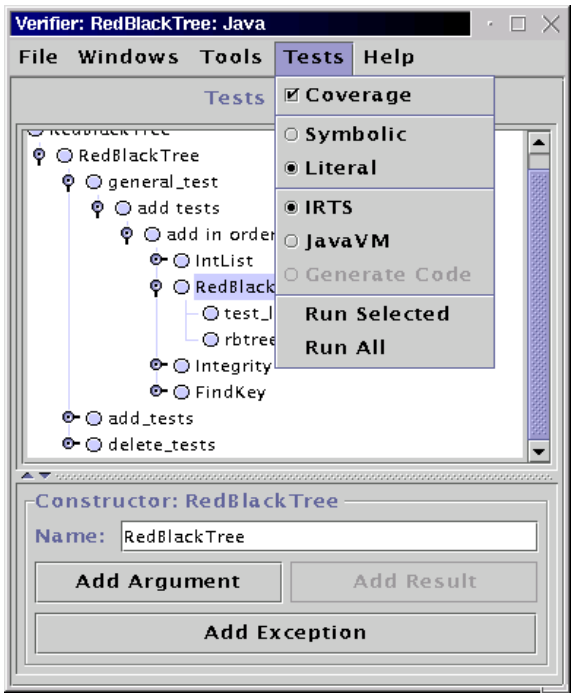

Figure 5. CTB screen. The main window shows the structure of a test descriptor file: a user can select any part of the structure for adding new elements, editing existing ones or removing elements. An element editor (which knows the structure of the tree, *i.e.* the allowed attributes and child elements for the selected element) will appear to the right when editing. At this point the user has selected a test operation in a test set and is about to run it literally using the **IRTS**: coverage statistics have also been requested. "Run Selected" can be chosen at any level in the tree for a batch processing style - running all of the test operations in a test group, test set, interface or implementation. "Run All" runs all the tests for a component - after politely suggesting that the user might not chose to wait for an instant result.

A number of technical weaknesses have been identified with symbolic execution[8]. The most significant is its inability to work with methods that use certain types of array indexing. It also has difficulty dealing with ADTs that are not amenable to traditional algebraic manipulation - the most prominent example being the String type. Methods of overcoming some of these weaknesses, making symbolic execution more widely applicable, have been discovered and will be published elsewhere.

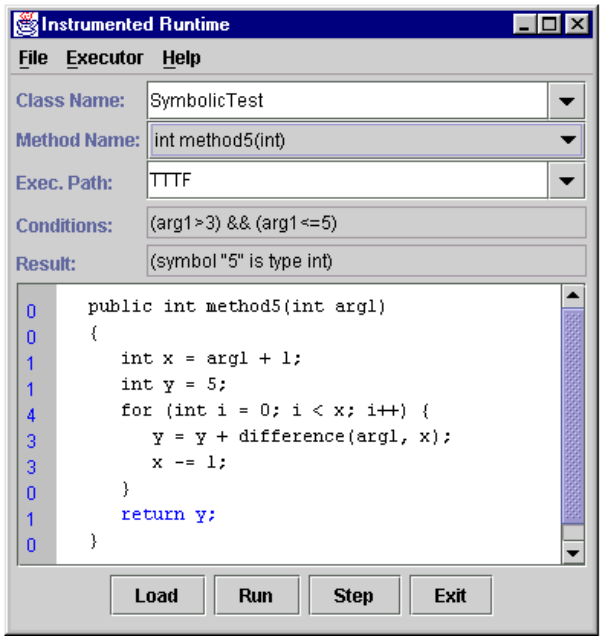

Figure 6. Symbolic execution and coverage. A tester has selected to test `method5` in class `SymbolicTest`. On the left of the window containing the method's source, the coverage counts may be seen: indicating that all statements in this method have been executed at least once. The condition field, which is generated by symbolic execution, shows a condition on the input that will run through the loop 3 times (indicated by the "`TTTF`" flags entered into the "Exec. Path" box). Thus the tester might include a test in which `arg1` is set to 4 producing a result of 5.

5.2. Literal Execution

Symbolic execution is necessarily slow! The literal execution mode is used when it is only desired to check or collect results. For Java, it uses the standard virtual machine for efficient execution. The test specification is parsed and the reflection API used to locate the methods appearing in a test operation, invoke them and collect the results. For C or C++, a simple test program will be emitted, compiled and run. Results are collected in log files which are converted to `ResultSet` elements to add to the test descriptor.

6. Regression testing

A key function of a testing tool is the management of regression testing. When the **CTB** runs a test, it augments the XML test descriptor file with `ResultSet` elements (see figure 7) which contain actual results derived by running test operations. This part of the test descriptor contains a number of `ImpRes` elements with a sub-structure similar to the main test descriptor part with results for each possible interface in separate `IntfceRes` elements, which in turn contain `TestSetRes` elements, *etc.* Each `OpnRes` element is marked with the version of the implementation from which it was obtained, the date of successful tests and the date and output of unsuccessful tests. This provides a complete history of the results from various versions of the component - linked by version numbers and dates to a source code control system if one is used. For efficiency, a tester may specify a maximum number of errors to be captured to prevent the specification files being loaded with large amounts of essentially useless information in cases where the component contains gross errors. Discrepancies from expected (*i.e.* those classed as "specified") results are highlighted when tests are run. For automatically generated tests with "weak accept", "strong accept" or "pending" qualifiers, results obtained from previous test runs are compared with those from the current run and discrepancies highlighted. These are cases where a corrected component may have generated a correct result - and the previous version contained an undetected error. When advised of this type of discrepancy, the tester has uncovered a problem that requires further investigation and we would normally expect the result to be checked and upgraded to "strong accept" status.

```
<ResultSet>
  <ImpRes Name="Java">
    <IntfceRes Name="Java">
      <TestSetRes Name="general test">
        <TestGrpRes Name="Add sequence">
          <OpnRes SWVersion="0.1"
                  Date="9-6-2000 13:45">
            <Actual Name="found">true</Actual>
            <Actual Name="valid">true</Actual>
          </OpnRes>
        </TestGrpRes>
      </TestSetRes>
    </IntfceRes>
  </ImpRes>
</ResultSet>
```

Figure 7. `<ResultSet>` element of a test descriptor. This section is added to the test descriptor when the first test is run and augmented thereafter whenever new test results become available. See figure 1 for the overall document structure.

7. Discussion

Our **CTB** has been used to generate test specifications for several components available through our component

server, VeriLib[21]. Clients are able to download a package containing the component (compiled code and, optionally, source) as well as the XML test specification. The package includes a module from the **CTB** which runs the tests, compares output with the expected output and output from previous runs. Clients downloading components are able to immediately determine whether the component performs according to its specification in the target environment by running the test specification. They are also able to use the test specification to gain additional detailed information about the actual functions of the component as it is an XML document which can be read directly or - in the case of large documents - by using any one of a number of freely available XML editors to work through the structure selecting portions of interest.

8. Conclusion

If CBSE is going to deliver the promised benefits, then the components must be reliable. This implies that developers must be able to produce verified reliable components and users must be able to confirm the claimed reliability quickly and easily.

Our component test bench addresses both these needs: it provides a developer tool that allows multiple ways to specify tests and captures the test specifications in a portable, readable way. It captures test results and the source of the result (*i.e.* derived directly from specification or automatically generated) so that regression testing can be rapidly carried out and so that problems which may have lain dormant for some time are highlighted for the developer's attention. By providing a history of performance - linked to software version numbers - it also provides the equivalent of a source code control system which monitors result changes rather than code changes.

We have observed a wide variety in the style of testing required by different components. Thus we added the concept of a test operation which enables a rich variety of test environments to be created in a standard way. To further cater for a diverse range of component testing requirements, we have prescribed a number of different ways of generating the test specification: enabling the developer to choose a style that suits the component under test or personal preference. Allowing predecessor test operations makes a significant contribution to testing productivity by saving much repetitive work.

Furthermore, by using XML - a standard document format which is rapidly gaining acceptance in a variety of environments - for the test specifications, we create the possibility of portable test specifications that are easily interpreted and executed by verifier tools other than our verifier.

In critical environments, developers cannot be expected to trust unknown components, thus the test specifications are as important as the component itself and must be equally portable. Our system allows a test specification along with a lightweight, portable tool - the test pattern verifier module - to run the tests to be packaged with a component. Thus users are able to rapidly confirm that a component runs in the target environment exactly as required and, further, that when updates are received, that they too conform to the required specification. This enables them to gain the expected benefits of CBSE through trust in the building blocks of the system they are constructing. Naturally, by using a single, standard test specification - rather than custom test programs - a developer need not download the tool every time a small component is imported, any existing tool which can read the specification may be used, saving considerable time if the testing tool is already familiar.

Further work

We plan to extend the current work in a number of ways: by providing closer integration with visual modelling tools, and by using the **IRTS**' internal representation of the source code to identify definition-use paths, we will provide further automation of the test selection process.

9. Acknowledgments

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

References

[1] M. Aoyama. New age of software development: How component-based software engineering changes the way of software development. In *Proceedings: 20th International Conference on Software Engineering*, pages 30–37. IEEE Computer Society Press / ACM Press, 1998.

[2] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[3] D. Box. *Essential COM*. Addison-Wesley, 1999.

[4] A. W. Brown and K. C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, sept/oct 1998.

[5] G. A. Bundell, G. Lee, J. Morris, S. Hope, S. Parr, and R. G. Dromey. *Component Software: A White Paper: Part II. Technical Aspects*. Software Engineering Australia (WA): `http://ciips.ee.uwa.edu.au/Research/SCL/white6.pdf`, 2000.

[6] H. Buwalda. *Testing with Action Words*, chapter 22. Software Test Automation: Effective use of test execution tools. Addison-Wesley, 1999.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[8] P. D. Coward. Symbolic execution and testing. *Jnl Information and Software Technology*, 33(1):53–64, Feb 1991.

[9] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, Apr 1994.

[10] R. Ferguson and B. Korel. The chaining approach for software test data generation. *IEEE Transactions on Software Engineering and Methodology*, 5(1):63–86, jan 1996.

[11] M. Fewster and D. Graham. *Software Test Automation: Effective use of test execution tools*. Addison-Wesley, 1999.

[12] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings: 17th International Conference on Software Engineering*, pages 179–185. IEEE Computer Society Press / ACM Press, 1995.

[13] M. G. Hinchey and J. P. Bowen. To formalize or not to formalize? *IEEE Computer*, 29(4):18–19, Apr 1996. In H. Saiedian, editor, *An Invitation to Formal Methods*, pages 16–30.

[14] ICSE. *Ensuring Successful COTS Development*. ICSE COTS Workshop Summary: `http://wwwsel.iit.nrc.ca/projects/cots/icsewkshp/mainsummary.html`, 1999.

[15] D. Megginson. *SAX 2.0: The Simple API for XML*. `http://www.megginson.com/SAX/`, 2000.

[16] U. S. Navy. *Virginia Class Submarine Program*. `http://www.chinfo.navy.mil/navpalib/ships/submarines/centennial/subinno%.html`, 2000.

[17] J. A. Profdeta, N. P. Andrianos, B. Yu, B. W. Johnson, T. A. DeLong, D. Guaspari, and D. Jamsek. Safety-critical system built with COTS. *IEEE Computer*, 29(11):54–60, Nov 1996.

[18] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, Apr 1985.

[19] J. Siegel. *CORBA: Fundamentals and Programming*. John Wiley, 1996.

[20] Software Component Laboratory. *Component Test Bench Documents*. `http://ciips.ee.uwa.edu.au/Research/SCL/Docs.html`, 2000.

[21] Software Component Laboratory. *VeriLib: A Source of Reliable Components*. `http://www.verilib.sea.net.au`, 2000.

[22] A. Thomas. *Enterprise JavaBeans: Server Component Model for Java, White paper*. `http://www.javasoft.com/products/ejb/`, Dec 1997.

[23] M. R. Vigder, W. M. Gentleman, and J. Dean. *COTS Software Integration: State of the Art*. NRC-CNRC Technical Reports, Software Engineering Group, Jan 1996.

[24] E. J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, Sep 1998.

[25] J. C. Wileden. Programming languages and software engineering: Past, present and future. *ACM Computing Surveys*, 28(4es):202, Dec 1996.