

Software Verification and Functional Testing with XML Documentation

Ernest J. Friedman-Hill
Distributed Systems Research Department, MS 9012
Sandia National Laboratories
Livermore, CA 94550
ejfried@ca.sandia.gov

Abstract

Continuous testing is an important aspect of achieving quality during rapid software development. By making the user documentation for a software product into part of its testing machinery, we can leverage each to benefit the other. The documentation itself can be automatically tested and kept in a state of synchronization with the software. Conversely, if the documentation can be machine interpreted, evaluation of the software's adherence to this description simultaneously verifies the documentation and serves as a functional test of the software.

This paper presents an application of these ideas to a real project, the manual for Jess, the Java Expert System Shell. The Jess manual is rich in machine-interpretable information and is used in several distinct modes within Jess' extensive functional and unit test suites. The effort to maintain the accuracy and completeness of Jess's documentation has dropped significantly since this method was put in place.

1 Introduction

1.1 Extreme Programming

In today's Internet economy, time-to-market has become of paramount importance in software development. New methodologies for rapid software construction such as Extreme Programming (XP)[3] are making it possible to deliver high-quality software faster than ever before. XP is a development model in which software is released to customers "early and often," via a series of incremental releases. Changing requirements are incorporated at each iteration, thereby putting useful software in the hands of users as soon as possible.

XP's four cornerstones are communication, simplicity, feedback, and courage. The XP method emphasizes *communication* between developers and customers, and

amongst developers. It urges *simplicity* in design and implementation. XP developers get *feedback* both from customers at the earliest possible stage, and from a rigorous battery of unit tests. Given this, XP programmers have the *courage* to work in today's continuously changing environment.

1.2 XP and Testing

As mentioned above, then, testing is an important aspect of XP. Ideally, a test is written for a feature before that feature is implemented. The tests are run before and after any change is made. System integration is done continuously; this is possible because the testing regimen immediately gives feedback about the compatibility of new features and old.

XP tests must run quickly and automatically, and must provide clear and immediate indication of success or failure. If the tests require too much programmer time or effort to compose or to run, the benefit of immediate feedback is lost, and the tests generally won't be used as they should be. In particular, XP uses two traditional kinds of tests: unit tests[10], targeted at specific routines or classes, and functional tests[10], designed to verify the overall correctness of an integrated system.

1.3 The Documentation Problem

There are, however, some difficulties associated with the XP development scheme. One problem is that the emphasis on change makes writing and maintaining accurate end-user and client-programmer documentation harder, especially for small projects and open-source projects. Documentation is already a sore spot for these kinds of projects, as described in the following (all too familiar) plaint:

Sigh.

In the proprietary software world, the marketing weenies are in charge, and the documentation and sales propaganda precede the reality; hence the term vaporware. In the free software world, the nerds rule, and the documentation often trails the reality[4].

The problem is simply that writing documentation takes time away from development, and so it is often left until last. This means that early releases will arrive without user documentation - an unfortunate consequence of an otherwise praiseworthy technique for rapidly delivering quality software.

One solution to this problem would be to make the user documentation into part of the test suite. If the documentation were interpretable by computer, then the behaviour described in the documentation could be verified by the test machinery. Writing documentation would no longer be a “superfluous” activity, but instead it would be an integral part of the development process. Inaccurate documentation becomes as serious as any other bug detected during testing.

We have applied this technique to a real project, the ongoing development of Jess[2], the Java Expert System Shell, using XML[1] as the documentation format. This paper describes this effort and suggests some potential enhancements for future work.

2 Jess

Jess[2], the Java Expert System Shell, is basically an interpreter, written in Java, for an idiosyncratic version of LISP (*the Jess language*.) Jess can be used in two overlapping ways. First, it can be a rule engine - a special kind of program that very efficiently applies *rules* (basically if-then statements) to data. A rule-based program can have hundreds or even thousands of rules, and Jess will continually apply them to data in the form of a *knowledge base* (symbolic facts which form antecedents to rules.) Often the rules will represent the heuristic knowledge of a human expert in some domain, and the knowledge base will represent the state of an evolving situation (an interview, an emergency). In this case, they are said to constitute an *expert system*. Expert systems are widely used in many application domains. Among the newest applications of expert systems are as the reasoning part of “intelligent agents”, in enterprise resource planning (ERP) systems, and in order validation for electronic commerce.

But the Jess language is also a general-purpose programming language, and furthermore, it can directly access all Java classes and libraries. For this reason, Jess is also frequently used as a dynamic scripting or rapid application development environment. While Java code gener-

ally must be compiled before it can be run, a line of Jess code is executed immediately upon being typed. This allows one to experiment with Java APIs interactively, and build up large programs incrementally. It is also very easy to extend the Jess language with new commands written in Java or in Jess itself, and so the Jess language can be customized for specific applications.

Jess is therefore used in a range of different ways, meaning that its documentation must cover many topics. The software is in use at hundreds of sites around the world in industries including e-commerce, insurance sales, telecommunications, and R&D, so the documentation must be of sufficient quality and completeness to satisfy the broad user base.

The Jess project is primarily a research project. While the basic syntax of the Jess language stays relatively constant, features are added and removed on a regular basis as requirements evolve and new ideas are tried out. Nevertheless, Jess is a small project, supported by one person working part-time. Taken together, the small project size, the dynamic nature of the software itself, and the large user base make the problem of maintaining up-to-date documentation for Jess particularly acute.

2.1 Jess’s Manual

The Jess manual has to accomplish several tasks; it must

1. Explain rule-based systems
2. Document the syntax of the Jess language
3. Document all the functions provided by the Jess language
4. Document Jess’ Java APIs

Whereas the sections explaining rule-based systems will rarely change, the other parts of the manual are subject to frequent and sometimes dramatic updating as Jess evolves. The manual includes source code examples in both the Java and Jess languages, and transcripts of interactive sessions with the Jess language interpreter. Currently the manual runs to about 150 pages of single-spaced formatted text (not including machine-generated documentation for the Java APIs,) the size of a small book.

2.2 Jess’s Test Suite

The test suite used in Jess’s continued development currently consists of more than 50 modules; each module performs dozens of tests. The suite is evenly divided between regression and unit tests. The unit tests use Beck and

```
<?xml version="1.0"
  encoding="US-ASCII"?>
<!ELEMENT paragraph (#PCDATA)>
<!ELEMENT document (paragraph+)>
<!ATTLIST document
  title CDATA REQUIRED>
<!ATTLIST paragraph topic CDATA "">
```

Figure 1: A very simple Document Type Definition.

Gamma's JUnit[11][12] as a harness, while the regression tests use a script-driven harness developed in-house. Many of the regression tests incorporate information from Jess' documentation.

3 XML and XSL

3.1 XML

The eXtensible Markup Language (XML) is an emerging text-based standard for structured data interchange[1]. The standard is owned by the World Wide Web Consortium (W3C,) an industry group with more than 400 member organizations that works to develop common protocols for the Web and the Internet. Work on the XML standard began in 1996, and XML 1.0 was approved in 1998. Numerous ancillary standards or proposed standards are also under the stewardship of the W3C. Both commercial and open source tools for working with XML are now widely available, and XML is seeing extensive use, especially in e-commerce applications.

XML has the familiar syntax of all languages derived from SGML: pairs of *tags* delimited by matching '*i*' and '*i*' characters, sometimes containing optional name-value pairs called *attributes*. This structure is of course very similar to that of HTML, and so XML can easily represent the contents of an HTML document. In fact, it is possible to define HTML in terms of XML, such that any well-formed HTML document can also be interpreted as XML.

XML's syntax is somewhat stricter than HTML's, however. Whereas in HTML, quotation marks are optional around attribute values, they are required in XML. Closing tags are also always required in XML, but are optional in HTML. Most importantly for our purposes, an XML document must strictly follow a Document Type Definition (DTD) which specifies the allowed tags, their possible relationships, and their allowed attributes. This means that an XML document can be easily parsed, manipulated, and verified, making it an ideal candidate for a testing documentation language.

Figure 2 shows a simple XML document, and Figure 1 is the DTD that governs its structure. The ELEMENT en-

```
<?xml version="1.0"
  encoding="US-ASCII"?>
<!DOCTYPE doc SYSTEM "simple.dtd">
<document
  title="Polar Animals">
  <paragraph topic="penguins">
    The penguin is a
    curious bird...
  </paragraph>
  <paragraph topic="polar bears">
    Polar bears are
    massive creatures.
  </paragraph>
</document>
```

Figure 2: A simple XML document following the DTD in Figure 1.

tries in the DTD correspond to allowed tags in the XML (an *element* is a pair of tags together with the text they enclose.) The allowed combinations and containment relationships among the tags are also specified by the ELEMENT entries using a notation reminiscent of regular expressions; for example, the document tag in our example can contain one or more paragraph tags. The notation #PCDATA means *parsed character data*, or simply text. The ATTRIBUTE entries indicate the name-value pairs that can be attached to an element. In our example, documents must have a title attribute, and paragraphs can optionally specify a topic.

The most important thing to note about this example is that not all of the information is intended for use in a printed version of the document. The topic paragraph attribute is instead intended for use by electronic document indexing and retrieval software. This technique of annotation with additional information is used in Jess's documentation.

3.2 XSL

XSL, the eXtensible Style Language, is a set of specifications for tools that can be used to format or interpret XML documents[5]. Of particular interest here is the XML Transformation rules (XSLT) specification[6], which can be used to write simple scripts that describe a transformation between XML and another format; in particular, XSLT can be used to describe a mapping between a particular XML DTD and HTML for presentation on the Web. Figure 3 is an XSLT script that translates our simple XML documents into valid HTML; the result is shown in Figure 4. Note how the script emits specific HTML tags for each of the XML tags and attributes in our DTD.

In our work, the original documentation is written in

```

<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl=
    "http://www.w3.org/XSL/Transform/1.0"
  xmlns=
    "http://www.w3.org/TR/REC-html40"
  result-ns="" >
<xsl:template match="document">
  <head>
    <title>
      <xsl:value-of select="@title"/>
    </title>
  </head>
  <body>
    <xsl:apply-templates/>
  </body>
</xsl:template>

<xsl:template match="paragraph">
  <p>
    <xsl:apply-templates/>
  </p>
</xsl:template>
</xsl:stylesheet>

```

Figure 3: A simple XSLT script following the DTD in Figure 1.

```

<html>
  <head>
    <title>Polar Animals</title>
  </head>
  <body>
    <p>
      The penguin is a
      curious bird...
    </p>
    <p>
      Polar bears are
      massive creatures.
    </p>
  </body>
</html>

```

Figure 4: The result of applying the XSLT script in Figure 3 to the XML in Figure 2.

XML. Two different XSLT translation scripts are employed to prepare user documentation. One is used to transform the XML into an online HTML manual for Jess. The other transforms the XML into new XML documents conforming to the DTD used by the TeXML translator[9], which generates \TeX source from which a printable PostScript manual is prepared.

3.3 Writing Documentation with XML

Although graphical XML editors exist, the XML source for the Jess manual was written by hand and is maintained by hand using the GNU Emacs editor and its built-in SGML mode. The tools for processing the XML were written using IBM's xml4j[7] XML parser and LotusXSL XSLT translator[8]. In practice, we have found that writing documents in XML format is no more difficult than writing documents of comparable complexity using LaTeX.

4 Functional Testing with XML

Software documentation normally has two tasks: it explains the need for the software, and it explains how to use it through prose and examples. For the purpose of this paper, only the second task is interesting. Our first goal is to ensure that all of the examples in the documentation represent the actual behaviour of the software. Our second goal, to the extent possible, is to confirm that the prose does as well.

To make the documentation for a software product part of its testing harness, it is necessary to encode the document's information about the software's expected behaviour in a machine-readable way. There are potentially many kinds of information that might be included:

1. For a code library, usage examples. These could include sample code to be compiled and expected output.
2. For a program that accepts scripts or textual input, sample inputs and expected outputs.
3. For either a code library or an interpreter, documentation for individual functions.
4. For a program with a graphical interface, descriptions of actions taken and expected results.
5. For an embedded system, example sensor inputs and effector outputs.

Jess is, as previously mentioned, a programmer's library containing an interpreter for a script language;

```
<?xml version="1.0"
  encoding="US-ASCII"?>
<!ELEMENT java (class,result?)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT result (#PCDATA)>
<!ATTLIST class
  filename CDATA REQUIRED
  noexecute CDATA ""
  embed CDATA "">
```

Figure 5: The parts of the DTD describing the <java> tag.

therefore, items 1, 2, and 3 in the list above are relevant and results will be presented here. We will also consider possible implementations for some of the other items in the list.

The actual testing activity is carried out by a dedicated program which parses the documentation (using a third-party XML parser,) verifies that it is well-formed according to the DTD, and then performs the tests. Each test generally involves finding all the instances of a specific XML tag in a document, then performing some action for each instance. The activity to be performed varies according to the particular tag; each tag type corresponds to one of the cases enumerated above. In the following sections, we will consider the design and application of tag types for each of these cases.

4.1 Testing Java Code Samples

Code examples in software documentation should always be tested for proper compilation; they can also be executed and, optionally, the result verified against a sample output. Sometimes an example should be compiled, but not executed (if it would have destructive results, for example.) The part of our DTD that represents Java code examples must therefore allow the documentation writer to specify each of these options. The part of the Jess manual DTD relevant to testing Java code snippets is shown in Figure 5.

A java XML element is made up of a single class element and optionally a single result element. The class element can specify filename, noexecute, and embed attributes. When the documentation is being used for testing, the text of the class element is extracted and compiled. If noexecute is not specified, the compiled class is executed and any output collected. If a result element is specified, the output is compared to its text. A test succeeds if none of these stages results in an error.

Note that the default is to execute the sample and collect the output. It might be argued that for the sake of

```
<java>
<class filename="ExABC">
import jess.*;
public class ExABC
{
  public static void
  main(String[] unused)
  throws JessException
  {
    ValueVector vv =
      new ValueVector();
    vv.add(new Value("a", RU.ATOM));
    vv.add(new Value("b", RU.ATOM));
    vv.add(new Value("c", RU.ATOM));
    // Prints "(a b c)"
    String s = vv.toStrinWithParens();
    System.out.println(s);
  }
}
</class>
<result>
(a b c)
</result>
</java>
```

Figure 6: A <java> tag as written in XML.

```
import jess.*;
public class ExABC
{
  public static void
  main(String[] unused)
  throws JessException
  {
    ValueVector vv = new ValueVector();
    vv.add(new Value("a", RU.ATOM));
    vv.add(new Value("b", RU.ATOM));
    vv.add(new Value("c", RU.ATOM));

    // Prints "(a b c)"
    String s = vv.toStrinWithParens();
    System.out.println(s);
  }
}

C:\> java ExABC
(a b c)
```

Figure 7: How the <java> tag in Figure 6 is rendered in printed documentation.

safety, the default should be to not execute each sample unless specifically requested. However, experience demonstrated here that almost all samples are executed and use of noexecute is rare, so the current default is useful.

Each stage of such a test proves something about the software and about the documentation. Simply compiling the example verifies the syntax of the documentation, while simultaneously proving that the software supports the API described. Executing the example without error demonstrates that it is free of gross semantic error, and comparing the output against the example is obviously tantamount to verifying this one aspect of the software's adherence to a specification, while simultaneously verifying that the user can do what the documentation claims, and will see the promised results.

A simple example of a `<java>` XML element is shown in Figure 6. During testing, the Java class `Test` is compiled and executed and its output compared to the given sample. The `filename` attribute of the `class` element is used to name the Java source file during the compilation process. In the printed documentation, this XML appears as in Figure 7. Here, the same `filename` attribute is used to form the example command line.

One complication is that in Java, the smallest compilable unit of code is the single complete class, so it is either necessary to require that all code examples be in the form of a complete class, or to define a standard environment in which a snippet can be embedded if required. We have defined such an environment appropriate for testing Jess. If a `class` element specifies a value for the attribute `embed`, then the code for that class is included as the body of a `main` method in a class called `Test`. For convenience, an `import jess.*;` statement is included in this skeleton file.

4.2 Testing Jess Code Samples

Testing scripting language code examples is similar to testing compiled examples, but it can also raise additional issues. The most important of these is that a single example usually consists of an interactive session including a series of inputs and a series of responses. In addition, a series of scripting examples will frequently be written such that each depends on the state left behind by previous examples, despite intervening blocks of manual text. The smallest executable block of code is typically a single line or function call, and so the environment in which the code is executed is necessarily more elaborate. Figure 8 shows the sections of the Jess DTD relevant to testing Jess language code snippets. Note that the `result` element is the same one as is used in Figure 5.

There are two important ways in which the DTD frag-

```
<?xml version="1.0"
      encoding="US-ASCII"?>
<!ELEMENT jess (input,result?)*>
<!ELEMENT result (#PCDATA)>
<!ELEMENT input (#PCDATA)>
<!ATTLIST jess noclear CDATA "">
```

Figure 8: The parts of the DTD describing the `<jess>` tag.

```
<jess>
  <input>(defglobal ?*frame* =
    (new java.awt.Frame
      "Frame Demo"))</input>
  <result>TRUE</result>

  <input>; Directly call 'isVisible'
  (printout t
    (call ?*frame* isVisible)
    crlf)</input>
  <result>FALSE</result>

  <input>; query the property
  (printout t
    (get ?*frame* visible) crlf)</input>
  <result>FALSE</result>
</jess>
```

Figure 9: A `<jess>` tag written in XML.

```
Jess> (defglobal ?*frame* =
  (new java.awt.Frame "Frame Demo"))
TRUE
Jess> ;; Directly call 'isVisible'
  (printout t
    (call ?*frame* isVisible) crlf)
FALSE
Jess> ;; query the property
  (printout t
    (get ?*frame* visible) crlf)
FALSE
```

Figure 10: How the `<jess>` tag in Figure 9 is rendered in printed documentation.

ment in Figure 8 differs from the one in Figure 5. In the `<jess>` DTD, a `<jess>` element can contain any number of inputs, each one optionally followed by a result. This allows a single `<jess>` element to capture an entire interactive session with the language interpreter. The second difference is denoted by the presence of the `noclear` attribute. Using this attribute, multiple `<jess>` tags separated by body text in the manual can share a single interactive session, so that each example can build on the state left behind by its predecessors. By default, the test harness will clear the previous environment; if the `noclear` tag is present, it will not.

Figure 9 shows what a typical `<jess>` tag looks like in XML. Figure 10 shows how this tag is rendered in HTML.

4.3 Documenting Individual Functions

A manual documenting a scripting language must document every function present in the language, and must not document any functions which are not present. While this sounds obvious, maintaining this can be difficult when the number of functions is large, and functions are added and removed frequently.

To help verify that the documentation and software match in this regard, the Jess manual uses a `<functiondef>` element. This element includes subelements that describe the arguments, return value, and location of each function. At the present time, this information is in English, not in machine-readable form, so it is used only in preparation of the printed manual. The list of all `<functiondef>` elements is, however, validated against the list of all available Jess language functions by the test suite (the Jess function `(list-function$)` computes this list.)

4.3.1 Testing the Test Suite

Although the argument and return value information from the `<functiondef>` tags are currently not validated, the manual-driven tests are not the only ones in Jess' test suite. There should be regression test coverage for every function in the script language in traditional hand-written tests. The `<functiondef>` tags are used to verify this. The full regression test suite is searched for at least one invocation of every documented function; this test fails unless all are found.

4.4 Testing GUI Actions

Applying these ideas to testing graphical interfaces is somewhat more involved than testing the script-driven applications we've discussed so far. Neither the inputs nor the results will be textual, so that a (possibly separate)

representation language will be needed for both. Nevertheless, it should be possible to incorporate user documentation into the validation of some kinds of GUI-driven applications. Some of the requirements will be explored in this section.

4.4.1 A Unified Approach

Test-enabled documentation is used to generate two different representations of a set of user actions and results: one human-readable one for printing, and another machine-readable one for use by the test harness. For the text-mode applications we've discussed so far, it is easy to see how one XML document can be used to generate both representations. For graphical applications, it is somewhat more difficult to see how this could be accomplished. Indeed, one approach to the problem would be to include two separate representations together in the documentation. The problem with this approach should be obvious to any software engineer: multiple copies of information in a single document tend to become unsynchronized during maintenance. In the present case, this would make it impossible to trust test results, since one could not be sure if the printed document actually described the actions that were tested. Obviously, a unified approach is needed. Both the user actions and results should be described in XML in such a way that the testing specification and the printed documentation could both be generated from a single representation.

4.4.2 Representing User Inputs

For many kinds of user inputs — particularly button presses, menu selections, and the like — it is a simple matter to design a representation language from which both testing instructions and human-readable text (or diagrams) could be generated. For many applications, this would be enough: word processors, spreadsheets, viewers for various file formats, and so on.

Other applications requiring more general mouse input, however, could not be so easily handled. In documenting an example usage of a vector-graphics drawing program, for instance, it might be necessary to say "select the three circles by dragging over them." The testing harness would seemingly need a sophisticated pattern-recognition capability to perform this task. If this act were instead described in a language easier to interpret as machine commands — i.e., "drag from 10, 10 to 100, 100" — the pattern-recognition capability would now be required to render the document in English using the user's vocabulary.

One way out of this quandary might be to supplement the examples with graphics generated by the document-rendering software. For example, if the document source

contains commands equivalent to “drag from 10, 10 to 100, 100,” the generated documentation might read “drag to select the region shown in Figure 1,” and an accompanying figure would then show the user’s screen with a rubber-banded selection region enclosing the three circles.

4.4.3 Representing Results

The representation of results is in general a harder problem. For some applications (word processors and spreadsheets again come to mind) the effects of user actions are well-defined changes to a body of text, and so a straightforward textual representation scheme would be sufficient. A simple scheme could also be devised for describing the appearance of dialog boxes and other GUI entities and their responses to user actions (providing that the host operating system made it possible for the testing program to inspect the window hierarchy of the application).

For many applications, however, user actions result in changes to graphical entities that cannot be trivially described; an example is the effect of a “blur” tool in a painting program. Here the only effective representation might be a screen shot of the actual output window. The generated user documentation could then present the screen shot, while the test harness could compare it to a live screen capture of the application. It is not clear that this scheme would be convenient enough for the document writer to be worthwhile in practice.

5 Conclusions

The validation system described here proved itself to be very useful in the development process from Jess 4.0 to 5.1. The effort required to maintain good user documentation was greatly reduced. Approximately ten alpha and beta releases of Jess over the space of a year were made, and each shipped with a completely up-to-date manual. All of the examples in each of the manuals were correct; conversely, the software always performed as described in the manual.

Many extensions to this scheme are possible. The possibility for expanded use of `<functiondef>` has already been implied. If the argument and return-value descriptions were machine readable, then a series of simple tests for every documented function could be automatically generated to verify that the types and number of arguments, and the type and sometimes identity of the return value, adhered to the documentation.

Another possibility would be the confirmation of the existence and signature of Java API functions mentioned in the manual. A special tag is already used to format such references in the printed documentation. Again, it should

be possible to automatically generate some very simple unit tests for such functions.

References

- [1] See <http://www.w3.org/XML/>.
- [2] See <http://herzberg.ca.sandia.gov/jess/>.
- [3] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000. Also see <http://www.extremeprogramming.com>.
- [4] Charles Curley, to the ntemacs-users@ca.washington.edu email list, March 2000.
- [5] <http://www.w3.org/Style/XSL/>.
- [6] See <http://www.w3.org/TR/xslt>.
- [7] See <http://www.alphaworks.ibm.com/tech/xml4j>.
- [8] See <http://www.alphaworks.ibm.com/tech/lotusxsl>.
- [9] See <http://www.alphaWorks.ibm.com/tech/texml>.
- [10] Bill Hetzel, *The Complete Guide to Software Testing*, (second edition,) QED Information Systems, 1988.
- [11] Erich Gamma and Kent Beck, “JUnit: A Cook’s Tour,” *Java Report* 4(5) May 1999, 27-38.
- [12] Erich Gamma and Kent Beck, “Test Infected: Programmers Love Writing Tests,” *Java Report* 5(7) Jul. 1998, 37-50.