

On the Use of Specification-Based Assertions as Test Oracles

David Coppit

Department of Computer Science
The College of William and Mary
Williamsburg, VA 23185
Email: david@coppit.org

Jennifer M. Haddox-Schatz

Daniel H. Wagner Associates, Inc.
Hampton, VA 23669
Email: jennifer@va.wagner.com

Abstract—The “oracle problem” is a well-known challenge for software testing. Without some means of automatically computing the correct answer for test cases, testers must use costly or untrustworthy techniques such as computing the results by hand, or using a previous version of the software. In this paper, we investigate the feasibility of revealing software faults by augmenting the code with complete, specification-based assertions. Our evaluation method is to (1) develop a formal specification of correct system behavior, (2) translate this specification into assertions, (3) inject or identify existing faults, and (4) for each version of the assertion-enhanced system containing a fault, execute it using a set of test inputs and check for assertion violations. Our goal is to determine whether specification-based assertions are a viable method of revealing faults, and to begin to assess the extent to which their cost-effectiveness can be improved. Our evaluation is based on two case studies involving real-world software systems. Our results indicate that specification-based assertions can effectively reveal faults, as long as they adversely affect the program state. Importantly, these faults are revealed without necessarily computing the expected output, thereby effectively sidestepping the oracle problem. In addition to these results, we describe techniques that we used for translating high-level specifications into code-level assertions. We also discuss the costs associated with the approach, and potential techniques for reducing these costs.

I. INTRODUCTION

A widely used approach for program verification is software testing. The primary goal of software testing is to identify faults in the software during development to help ensure that the software will not fail after deployment. A test case consists of an input and expected output, and the system is verified as correct for that input if the actual output matches the expected output. Despite the fact that testing provides a proof of correctness for only those test cases that pass, it remains popular due in part to its low, incremental cost.

A key challenge for testing is known as the “oracle problem”—the need for some trustworthy method of computing the correct expected output for test cases [1], [2]. A number of approaches have been proposed to address this problem, such as inferring partial specifications from the code or its behavior [3], [4], deriving oracles from specifications [5], or using specification-oriented analysis tools [6]. Such approaches are either costly to use, or have limited capability to reveal faults. As a result, testers usually rely on manual

computation of expected test outputs, which severely limits the number of test cases that can be developed, and precludes the use of automated testing.

In this paper we revisit the notion of self-checking programs. This is an attractive alternative to test oracles, based on the observation that checking a result is usually easier than computing one. For example, it is possible to check the correct operation of a sort algorithm in $O(n)$ time, compared to a computation that takes $O(n \log n)$ time. Algorithms for checking results are also usually simpler, making them easier to verify for correctness and therefore more trustworthy than oracle implementations. Self-checking programs also have other benefits: checks on internal state allow more precise location of faults, and can reveal invalid internal states that do not result in invalid output [7], [8].

The simplest form of self-check is the assertion. An assertion is a self-check on a portion of a running program’s data state [9]. Despite the seemingly widespread support for assertions [10]–[12], there are a number of key challenges to their effective use. First, developers are unsure of how to write effective assertions [12], and therefore assertions are added to programs in a haphazard manner, often as an afterthought. Such assertions provide poor verification of system behavior because they only address low-level details of the implementation, are woefully incomplete, and have little relationship to each other or the overall high-level specification.

To the authors’ knowledge, there has not yet been an assessment of the full potential of assertions to reveal faults. In particular, we do not yet know if it is possible to develop assertions that are sufficiently powerful to abrogate the need for test oracles. If we can establish the basic feasibility of using strong assertions to solve the oracle problem, then we can begin to search for more cost-effective variations. It is for this reason that, in this paper, we assume that cost is not a factor; we will use the lessons learned here to address cost-effectiveness in a future paper.

The approach we take for evaluating the full potential of assertions is as follows. First, we develop a formal specification of the software. We specify the behavior of the software in enough detail to provide a satisfactory definition of correctness. We then manually translate this specification in

its entirety into software assertions. Next, we use manual and automated fault injection to create a set of mutant programs. Finally, for each mutant, we execute a set of test cases on the self-checking code in order to determine whether the fault is revealed by the assertions.

We define a fault as a program error that causes the specification to be violated. In other words, we recognize that assertions are only as good as the specification from which they are derived. This approach defines the correctness of the software more appropriately in terms of the specification, instead of on the definition implied by programmer assertions, informal documentation, alternative implementations, etc.

To evaluate the feasibility of using specification-based assertions as test oracles, we applied our evaluation approach to two case studies. The first system, Auger, is a client/server application written in Java that provides an interface allowing users to submit batch jobs to a large server farm. The second system, the Test Driver Generator (TDoG) application, is a unit testing tool for Java. As is often the case, these systems were well documented but had no associated formal specifications. In addition, we had access to over 21,000 test inputs for Auger, but had to develop our own test suite for TDoG.

The primary contribution of this paper is an evaluation of the fault-detecting effectiveness of sophisticated assertions derived from formal specifications. In our two case studies, we found that specification-based assertions were very effective at identifying program faults. Of course, creating a formal specification of a system and embedding the assertions is costly. However, our experiences indicate that specification-based assertions can be a viable alternative to test oracles, and that further research is warranted to determine if the costs can be reduced without compromising effectiveness. We discuss potential ways to reduce the cost, and also discuss the process by which we converted the specification into assertions.

The rest of the paper is structured as follows. Section II provides a summary of related work. In Section III we present our evaluation approach using a simple example. In Sections IV and V we present our two case studies. In Section VI we discuss the process by which we translated specifications into assertions in more detail. Section VII provides an evaluation of our case studies and the approach. Conclusions and future work are given in Section VIII.

II. RELATED WORK

In this section we detail some previous work on self-checking software and the oracle problem. Space limitations prevent us from providing a complete treatment.

A. Self-Checking Software

A number of authors have argued and evaluated the benefits of using assertions as self-checks for the correctness of software [13]–[16]. More generally, several authors have suggested checking as an alternative to independent computation of expected results, and some have gone so far as to take corrective action [17]–[19]. In more recent work [20], [21], programs are monitored to ensure the correctness of temporal

logic properties. The performance issues with this approach limit the monitoring capability to partial specifications expressed in the form of constraints.

In general, assertion-based approaches ensure that only a limited set of properties hold for a given state of the system. Design by contract attempts to improve the rigor of assertions in object-oriented code [10], [11], [22] by providing more advanced programming language support for preconditions, postcondition, and class invariants. Originally pioneered in the Eiffel, design by contract has been extended to languages such as Java [23]–[25], Ada [26], and C++ [27].

To the best of the authors' knowledge, there has been little work on assessing the extent to which such technologies can be used to create effective assertions. Voas et. al address the issue of placement of assertions within a the code in order to increase the likelihood of uncovering faults [7]–[9]. Other work by Roseblum attempts to identify types of assertions that were most likely to reveal faults [12]. Roseblum suggests that it could be useful to derive assertions from formal requirements and high-level design specifications, but we are unaware of any subsequent research in this direction other than that which we present in this paper.

B. Solutions to the Oracle Problem

Traditional approaches to the oracle problem are to use a previous version of the software, reimplement a critical portion of the overall functionality in a trustworthy way, or compute the correct result by hand. Each of these approaches are unsatisfactory in that they are costly or are of questionable validity. Indeed, without an independent, documented specification, the only standard for correctness is an alternative implementation or the test suite itself.

A number of researchers have therefore turned to *specification-based testing*. Work by Richardson et. al [5] advocates the derivation of test oracles from formal specifications, since specifications accurately describe what the system is intended to do. More recent work by Boyapati et. al [28] seek to avoid the cost of creating by an oracle from the specification by instead translating inputs from the specification to the implementation, executing the implementation, and then translating the result back to the specification for checking against correctness constraints using a model checker. Their approach avoids the needs to generate an oracle, but sacrifices completeness—the program may or may not have a fault if the model checker does not report an error.

Other work has attempted to solve the oracle problem by inferring partial specifications. Work led by Michel Ernst [4] dynamically infers constraints on the values of variables in a program based on a training test set. These constraints can then be used to find faults using other test data, or even to assess the quality of the training test cases. For example, a *month* variable may be inferred as having a constraint of $0 \leq \textit{month} \leq 11$, but only if the test suite provides enough statistical confidence to allow the tool to determine that the constraint is not spurious. Work by Engler et. al [3] statically infers constraints based on how variables are used in a program. For example, a variable

that appears in the denominator of a division is inferred as being non-zero at that location in the program. Approaches such as these are lightweight in nature, but also result in inherently incomplete specifications. As a result, their fault-revealing effectiveness is reduced.

III. EVALUATION APPROACH

In this section we describe our approach for assessing the feasibility of using specification-based assertions to detect program faults.

A. Overview

1) *Creation of Specification-Based Assertions*: Ideally, the software for which the assertions need to be created would already have a corresponding formal specification that provides a definition of correctness. However, as is often the case, if such a specification does not exist we must instead develop one from the available documentation. To aid in the creation of a formal specification, it may be useful to first create an informal specification in English that reviews the system behavior in detail, and then convert this into a formal specification. The specification must then be validated as correct by domain experts.

Once a formal specification has been obtained or created, the next task is to convert it into assertions. To ease this task, it is better to write the specification using a formal language and structure that matches that of the software. For example, in this paper, we used Object Z [29], an object-oriented specification language, since the systems in our case studies were both object-oriented. Likewise, we structured the specification so that it was similar to the structure of the code.

For each corresponding specification and implementation module, we derive a set of assertions from the specification. Note that some predicates will be explicitly defined, while others, such as declarations, will be implicitly defined. Further, not all predicates must be turned into assertions. For example, a predicate stating that a data structure is finite is already guaranteed by a concrete implementation. Conversely, a predicate stating that a data structure is infinite will be impossible to satisfy. On the other hand, many low-level constraints on data types are handled by the exception mechanisms of modern programming languages, which helps to bridge this gap.

Once the assertions have been derived from the formal specification, the next step is to manually insert the assertions into the implementation. Ideally this step is performed with the use of an assertion specification tool that provides support for the creation of complex assertions involving quantifiers and other specification-oriented constructs.

Note that design by contract provides a compelling alternative to specification and translation process we have described. Programming languages that support design by contract allow us to directly express the specification of the system in the code itself. Care must be taken, however, to validate the specification independently of the code, so as to avoid incompleteness in the specification or implementation-oriented constraints. (A specification should describe *what* the software should do, but not prescribe *how* it should be done.

If the system used in the case study was developed without the use of a programming language that supports design by contract, then one must implement the assertions from a specification. There are a number of tools that enhance the simple assertion capabilities of most programming languages, adding higher-level specification features such as quantification. Examples for Java include Jass [24] and JML [23].

2) *Fault Injection* : In order to assess the fault-revealing effectiveness of the assertion-enhanced system, we require faults. Ideally, the system used in the case study would have a set of real faults that can be selectively enabled. However, in practice such systems are difficult to acquire, and for highly reliable software few faults may be available.

The approach we use in this paper is to use fault injection [30] to simulate faults in the system. Fault injection has had a rich history of success in assessing the reliability of hardware systems, and has had some success when used in software systems [31], [32]. Unfortunately, very little work has been done to develop techniques that inject faults, or even to assess the extent to which existing techniques emulate real faults. One study by Madeira et. al [33] found that only 56% of real faults could be feasibly simulated using fault injection.

There are two primary techniques for fault injection. One, called mutation testing, modifies the source code in ways that are intended to emulate programmer errors [34]. Examples of such manipulation include deleting a statement, reordering statements, and changing a $<$ to a $>$. An alternative approach is to perturb the values of variables in the software [35], [36]. This approach is both more abstract and more general, and is based on the notion that the ultimate effect of a fault is to adversely modify the runtime state of the software.

In order to perform fault injection on our test systems, we employed JavaWrap, a research prototype developed by Cigital Inc. to verify the correctness of third party Java components [37]. JavaWrap includes a fault injection subsystem that allows a user to specify which data types to be corrupted, as well positions in the code where corruption should occur. Additionally, the user can choose among several types of faults to inject. For example, if one wishes to perturb data of type `int`, one can specify that the data be incremented or decremented by some arbitrary value, assigned to a random value, set to zero, or set to the maximum or minimum value of the `int` data type.

Once the user has specified these items, the actual fault injection process is automated by JavaWrap at runtime. Because the fault injection occurs dynamically, the source code for the component to which fault injection is being applied is unnecessary. This is in contrast to mutation testing, in which the source code is manually manipulated.

3) *Testing* : Once the self-checking version of the software has been prepared, and we have identified a number of candidate faults, we can evaluate the effectiveness of the specification-based assertions. For each fault, we create a mutant of the system that includes only that fault. We then execute the tests, and determine whether an assertion is violated. If the assertions are effective, then any test input that

causes fault to adversely affect the program state should cause an assertion to fail.

B. A Simple Example

We now present an example to illustrate our evaluation approach. Consider the following pseudocode implementation of an algorithm to sort an array of Elements:

```
public void sort(Element a[], int size) {
  Element save;
  for (int i=1; i<size; i++) {
    save = a[i];
    int j;
    for (j=i; j>0 && (a[j-1]>save); j--)
      a[j]=a[j-1];
    a[j]=save;
  }
}
```

We begin by creating the following informal specification of the above algorithm.

- The elements of a at the end of `sort`'s execution must be some permutation of a 's elements at the start of `sort`'s execution.
- For each index i where i is in the range of zero to the size of a minus one, $a[i] \leq a[i+1]$ must hold.

At this point we pause to note that what is specified above is not unique to a particular sorting algorithm, such as insertion sort. In fact, this informal description and the formal specification that follows should hold for *any* sorting algorithm whose goal is to sort elements in nondecreasing order.

From this informal specification, we take the next step and create a formal specification. For example, we can formalize `sort` using the Z formal specification language [38] as follows:

[*Element*]

$0 \leq 1 \text{ : } Element \leftrightarrow Element$

$Sort : seq\ Element \rightarrow seq\ Element$

$\forall in, out : seq\ Element \bullet$
 $Sort(in) = out \Leftrightarrow$
 $items(in) = items(out) \wedge$
 $(\forall i : 1 \dots (\#out - 1) \bullet out(i) \leq_1 out(i + 1))$

In this specification, *Element* is declared to be a given type that represents an arbitrary item that can be stored in an array. We provide a type definition for the \leq operator for elements of the *Element* type, abstracting away the details. Finally, we define *Sort* as a function whose argument and result are both sequences of *Elements*. In the last two lines we formally express the informal constraints described previously. Note that we rely on the *items* function in this specification for *Sort*, which computes the bag of items in a sequence¹. The *items* expression ensures that the output sequence is a permutation of the input sequence.

¹A bag is an unordered set that may contain duplicate elements.

From this formal specification we derive several assertions. First, we have the assertion that the list of items must be finite, as sequences are finite in Z. This is an assertion that obviously does not have to appear in the implementation. Secondly, we can create an assertion corresponding to the predicate $items(in) = items(out)$. Assuming that the assertion language does not support *items*, we must create a helper function called `isPermutation()`. The resulting assertion is:

```
assert(isPermutation(old_a, a, size));
```

For the ordering predicate we can create a corresponding assertion embedded in a loop that iterates over the elements of the sorted list:

```
for (int i = 0; i<size-1; i++)
  assert(a[i]<=a[i+1]);
```

Now we embed the assertions into the implementation:

```
void sort(Element a[], int size) {
  // Save a copy of the list
  Element old_a[] = copy_array(a[],size);

  Element save;
  for (int i=1; i<size; i++) {
    save = a[i];
    int j;
    for (j=i; j>0 && (a[j-1]>save); j--)
      a[j]=a[j-1];
    a[j]=save;
  }

  // New assertions
  assert(isPermutation(old_a, a, size));
  for (int i = 0; i<size-1; i++)
    assert(a[i]<=a[i+1]);
}
```

In the interest of space we omit the implementation of the `isPermutation()` function. As the reader might suspect, implementing this helper function increases the chance of the assertion itself containing a fault. Later in the paper we will discuss the impact of weak language support on the cost of writing assertions.

Now that we have developed our specification-based assertions, we would like to determine whether they are effective at revealing faults that cause the function to operate incorrectly. Note, though, that our definition of correctness is in terms of the original specification. For example, the specification does not state whether the sort algorithm should be stable, whether it should have a particular algorithmic complexity, or even whether it must terminate. On the other hand, the resulting assertions are more powerful than the types of assertions that programmers typically use.

The approach we take at this point is to inject faults into the implementation, then execute the self-checking code for a number of test inputs. If the assertions are sufficiently powerful to reveal a fault any time an input results in the fault adversely modifying the program's state, then they can serve as a useful alternative to a separate oracle implementation.

IV. CASE STUDY #1: AUGER

A. Description of the System

Auger, the first case study system, was built to support the experimentation being performed by the physicists at Thomas Jefferson National Accelerator Facility (a.k.a. Jefferson Lab). Auger is an object-oriented application written in Java whose purpose is to provide a mechanism to allow the physicists to interact with the server farm and associated hardware components [39]. Auger implements a client/server model that allows a physicist to submit a job to the server via an input script. A job submission consists of the input files needed, the application to be run, and the location to which output will be placed. Auger’s *Job Server* transforms the request into a representation that can be understood by the underlying hardware components and schedules the thousands of submitted job requests in an efficient and fair manner [39].

The portion of Auger we selected for the experiment was the package containing the classes that encapsulate jobs and job submissions. This package contains the *JobSubmission* class, which stores the information for a job submission specified by a user input script. Each *JobSubmission* class contains general information about the submission, such as the user’s email address and the submission’s identification number, as well as the list of *Job* objects that comprise the submission. Each *Job* object can either be a *FileJob* or *WorkJob*. Every *JobSubmission* must contain at least one *WorkJob*, which represents a command to execute.

B. Creation of Specification-Based Assertions

Since the Auger system did not have an associated formal specification, the first step of our experiment involved creating a suitable specification based on the documentation and conversations with the developer. Since the system was implemented in standard Java, we chose not use design by contract. Instead, we used the Object-Z specification language to specify the job management system.

The resulting specification is approximately 13 pages long, including expository prose. We first developed an informal specification, then translated it into Object-Z. We validated the specification using a type checker, and by reviewing it line-by-line with the original developer.

Once we were satisfied that the specification provided a sufficiently complete definition of the correct operation of the software, we used it to derive a set of assertions. We expressed the assertions with the help of Jass [24]. Jass provides a somewhat sophisticated assertion mechanism, and operates as a preprocessor, translating Jass statements into standard Java. The job management package of Auger originally contained 834 lines of code, and we added 224 lines of specification-based assertion code to this package.

C. Fault Injection

In total, we used the JavaWrap tool to create 7 faults that were injected into the job classes; additionally, 5 faults were injected manually. Below is a list of the injected faults. Manually injected faults are indicated by a “*”.

TABLE I
AUGER TESTING RESULTS

Fault ID	Assertion Failures
1	21,529
2	21,529
3	21,529
4	21,529
5	21,529
6	21,529
7	17,403
8	10,479
9	10,479
10	10,479
11	10,423
12	4

TABLE II
TDOG TESTING RESULTS

Fault ID	Assertion Failures
1	10
2	10
3	16
4	22
5	25
6	3
7	22
8	10

- 1) Increment each *Job ID* number for a *JobSubmission* by 10.
- 2) Decrement the height of a *WorkJob* by 10 (the height stores the maximum length of the dependent jobs chain)
- 3) Decrement a *WorkJob*’s *ID* number by 10.
- 4) * Remove one *Job* object from a *JobSubmission*’s list of jobs
- 5) * Remove one *Job* object from a *JobSubmissions* list of jobs, and then adds a “dummy” *WorkJob* with an *ID* number of 10 to the list
- 6) * Add a job to the list of dependent jobs for the first *Job* object of the a *JobSubmission* object
- 7) Decrement the *JobSubmission ID* number by 10.
- 8) Decrement the value of the height variable for a *FileJob* by 10.
- 9) Decrement every *FileJob*’s *ID* number by 10
- 10) * Remove one element from every *Job* object’s dependency list (unless the dependency list is empty)
- 11) * Add a *WorkJob* to the dependency list of each *FileJob* if the dependency list is empty
- 12) Reassign the *String* object storing the job command to be a substring of the original

D. Testing

We were able to acquire 21,530 test inputs from Jefferson Lab, which were derived from real job submissions to the system. We first executed the specification-based assertion version with the test inputs and no faults. One test caused an assertion to fail. However, the violation was caused by a faulty test input script, as opposed to an inherent problem with the program itself. We removed this input from our test set.

Next, for each fault, we created a version of the software that contained the fault, and then re-ran the system for the set of inputs, recording all assertion violations. Table I contains our test results. The first column identifies the fault by its *ID* number from the above list of faults. The second column gives the number of test cases that failed. The assertions were able to detect the fault in all of the test cases for faults 1-6, and a large number of the test cases for faults 7-11.

Upon further analysis, we discovered that in the test cases that did not fail, the fault was not actually executed. That is, fault 7 only executed in cases where the input script specified

an email address, which was true for exactly 17,403 of the inputs. Faults 8-10 only executed in test cases that involved a `FileJob` object, and file jobs were specified in only 10,479 of the input scripts. Fault 11 only executed in test cases involving a `FileJob` whose dependency list was empty, and of the 10,479 test cases that would result in the creation of one or more `FileJob` objects, 10,423 input scripts satisfied this criteria. Thus, for faults 1-11, as long as the code containing the fault executed, one of our specification-based assertions identified it.

The specification-based assertions only performed poorly with fault 12. This fault involved reassigning a string variable by randomly selecting a character position in the `String` object, eliminating all characters before that position, and making the remaining characters the new, corrupted string. The resulting string was the empty string for several of the test cases, and when this was the case, an assertion stating that this variable could not be empty was violated. Otherwise no assertion violation occurred.

When we inspected the specification, we discovered that in these cases, the fault was not actually a fault. According to the specification, the string need only be nonempty to be valid. Since this is not true in practice, we should have more accurately specified this string, perhaps by providing a regular expression defining valid email address strings.

V. CASE STUDY #2: TDOG

A. Description of the System

The Test Driver Generator (TDoG) is a unit testing tool for Java [40]. Using this system, a user specifies a Java class to be tested and uses a GUI to specify which methods should be manipulated and in what order. Based on the user specification, TDoG then automatically generates a test driver which can be executed. Upon execution of the test driver, all output produced is processed by TDoG's output management system, the data recorder, which allows the state of arbitrary objects to be displayed in an organized manner. It is this subsystem that we selected for our second case study.

The data recorder takes an arbitrary object as input, recursively breaks it down into its primitive members, stores the decomposed state in a special data structure, and then displays this information in an organized manner for user observation. The central class in the data recorder subsystem is the `DataRecorder` class, which is responsible for decomposing an input object. The state of a decomposed object is stored in an `ExportObject`. The subsystem defines an abstract `ExportObject` class and several concrete implementations of this class. Each implementation is meant to store a certain type of object. For example, `PrimitiveExportObjects` store primitive values and `ArrayExportObjects` store the state of n-dimensional arrays.

B. Creation of Specification-Based Assertions

As was the case with the Auger system, TDoG did not have an associated formal specification. However, it was well-documented, and one of the authors (Haddox-Schatz)

was involved with its development. As before, we created a formal specification expressed in Object-Z. The specification is approximately 13 pages long, including expository prose. However, because the documentation for the software was fairly complete, we did not feel the need to first create a natural language specification.

This time, the amount of assertion code that we created as substantially smaller than the size of original system. The TDoG data recorder originally contained 3,475 lines of code, and we added 131 lines of specification-based assertion code.

C. Fault Injection

Because TDoG was a stable system, we were again in a position that required us to rely on fault injection. In total, we used the `JavaWrap` tool to create 4 faults that were injected into the data recorder classes; additionally, 4 faults were injected manually. Below is a list of the injected faults. Manually injected faults are indicated by a “*”.

- 1) In the method which recursively decomposes arrays, set the integer variable tracking the recursion depth to a negative value
- 2) In the `ArrayExportObject` class, set the data member that stores the number of dimensions in the array to a negative value
- 3) Set the string storing the name of a field in the decomposed object to the empty string
- 4) Set the unique hash code for the decomposed objects to a fixed negative value
- 5) * The `traverse()` method in the `DataRecorder` class decomposes an object given as input and stores its state in an `ExportObject`. At the end of the `traverse()` method, set the `ExportObject` attribute that stores the object's type to the empty string.
- 6) * In `RepeatedExportObject`, corrupt the value of the data member meant to store the unique ID for objects encountered more than once
- 7) * In `PrimitiveExportObject`, set the data member that stores the value of the primitive to null.
- 8) * Change the `traverse()` method so that it first checks whether or not the input object is a repeated reference rather than first checking whether or not it is null

D. Testing

Since we were not able to acquire any of the original test cases used for TDoG, we had to create our own test suite for the data recorder. Using JUnit, a unit testing tool for Java [41], we created 27 test cases for the data recorder. When creating these test cases, we considered the different types of objects that the data recorder expects as input and then included at least one example of each type in our test suite. To be certain that the data recorder would be thoroughly tested we simply needed to provide it with a variety of different types of data structures. Doing this ensured that it could properly deal with all input data regardless of the form in which it was stored. The actual values in the input object are irrelevant to the data recorder's functionality. Thus, the test inputs for the

data recorder included primitive types, one dimensional and multidimensional arrays, null objects, objects with primitive and non-primitive fields, objects with null fields, recursive data structures (e.g. circularly linked-lists), etc.

When we ran the version of the data recorder with the specification-based assertions and no faults, all of the test cases passed. Table II contains our results for each fault we injected. The first column identifies the fault by its ID number from the above list of faults. The second column gives the number of test cases out of 27 that failed in the specification-based assertion version of the data recorder.

We note that overall, the specification-derived assertions were effective at identifying the faults. As before, closer analysis revealed that for faults 1-7, those test cases which did not cause an assertion to be violated did not actually execute the injected fault. The only exception was fault 8, which only resulted in an assertion violation for 10 of the test cases. Upon further investigation, we discovered that these were the only test cases that involved a null value (and therefore were the only test cases that could reveal the fault). As a result, the fault must not only be executed, but it must also result in the program entering an invalid state.

VI. TRANSLATING SPECIFICATIONS TO ASSERTIONS

As a secondary contribution of this work, we describe issues we encountered related to the translation of specifications into assertions. We related these issues in this section.

A. The Specification/Implementation Gap

Since a formal specification deals with the high level view of a system, it does not address certain low level concerns that should be considered when writing assertions. For example, it might be prudent to check that a list object is not null before invoking a method to add an element to the list. However, in most formal specification languages, the concept of a null object does not exist. Thus, when making the translation to assertions, it is important to be aware of this gap that exists between specification and program code and account for it when writing the assertions.

Another example of the gap that exists between specification and code is the notion of size constraints on data types. There are limits to the maximum and minimum values that data types such as `int`, `float`, etc. can take during program execution, otherwise underflow or overflow can occur. In a specification, these issues are abstracted away. Thus, overflow and underflow are of no concern in a specification and so we cannot count on specification-based assertions to identify potential errors related to size constraints.

We also found that it was very useful to structure the specification in order to minimize “structural gaps” between the specification and implementation. This allowed us to more directly map specifications to their proper locations in the specification. We also found that choosing a specification language with a similar semantics eases the translation process.

```
public abstract class Job { ... }
public class WorkJob extends Job { ... }
public class FileJob extends Job { ... }
public class JobSubmission {
    private List m_jobs;
    private String script;
    ....
    /** invariant forall i: {0 .. m_jobs.size()-1}
        # ((Job)m_jobs.get(i) instanceof WorkJob ||
        (Job)m_jobs.get(i) instanceof FileJob) &&
        script != null
    */
}
```

Fig. 1. Example of Class Invariants

B. Types of Assertions

As we applied our approach we noticed that certain patterns emerged during the translation process. In general, we noted that we were either deriving assertions to check the correctness of the execution of a complex algorithm or to check that information stored in a data structure adhered to specific constraints. Thus, we can classify the assertions we derived into two broad categories: algorithm checking assertions and data structure checking assertions. For algorithm checking assertions, a set of post condition assertions would be placed after the body of the algorithm to ensure that it executed properly. Any supporting functions that the algorithm calls could have precondition and postcondition assertions as well, if they were specified. That is, these preconditions and postconditions would be responsible for performing “localized” checks related to that method’s purpose, and perhaps verify intermediate steps of the algorithm as well.

We noted that placing an assertion to check the overall correctness of the algorithm’s outcome in one of the supporting functions could result in a false sense of security of the implementation’s correctness. Though it may be useful to know that such a condition is true during the algorithm’s execution, what matters is whether or not that condition is true when the implementation of the algorithm produces its final result(s). Also, we noticed little importance for class invariant style assertions when verifying the overall correctness of a particular algorithm.

For data structure checking assertions, we realized that as we translated specifications into assertion code, that invariant assertions were of most use. In fact we actually discovered that invariant assertions can be split into two categories: an invariant that applies to a data structure at any time and an invariant that applies to a data structure only after that data structure has been fully populated. This discovery came about due the very detailed description of the format of certain data structures provided by the specification.

To demonstrate the use of these two categories of invariants, consider the abbreviated versions of the Auger job classes as shown in Figure 1. The Auger job package defines an abstract `Job` class, and then two subclasses defining the two types of jobs that can be submitted for execution: work jobs and

file jobs. The `JobSubmission` class encapsulates all jobs and associated information in the job submission input script. The `m_jobs` variable stores the jobs and the `script` variable stores the input script the user submitted. The invariant is written using the Jass assertion language. The first clause states that each job in the job list must be of type `WorkJob` or `FileJob`—it can't simply be of the concrete base class type `Job`. The second clause checks whether or not the input script variable is null. From the time a `JobSubmission` object is instantiated, the first clause should *always* be true; any time that a job is added to the list its type must be that of `WorkJob` or `FileJob`. This can not be said for the second clause, as the `script` field is populated some time after the `JobSubmission` constructor finishes execution. Thus, leaving the second clause in the invariant will result in the invariant always being violated. Taking the clause out omits an important constraint that should be satisfied sometime in the future: every `JobSubmission` object must be assigned a corresponding input script.

C. Assertion Design Patterns

Since neither Jass nor any other assertion tool of which we are aware provides support for differentiating between these two types of invariants, we created a new design pattern, called the *object coherence pattern* to support invariant assertions that should apply to a fully populated object. This coherence pattern provides a way to indicate when an object has been fully populated, resulting in a way to check conditions that must true of a fully populated object. This pattern requires that an additional boolean variable, `isCoherent` be added to the class as well as an `makeCoherent()` method. The boolean variable is initially set to false. It appears in the invariant assertion as follows:

```
/** invariant !isCoherent ||
    <assertions applying to completed object> */
```

The `makeCoherent()` method simply sets the boolean variable to true. It is the responsibility of the code that constructs the given object to call this method to indicate that the object has been fully populated. If this code has constructed the object correctly, then calling the `makeCoherent` method should not result in an assertion violation. In Figure 2, we show how the object coherence pattern is applied to the `JobSubmission` class.

Applying this new pattern ensures that the `script != null` assertion is not checked until after the calling program has had a chance to completely populate the object. Once the `makeCoherent()` method has been called, this assertion will be checked upon all future invocations of `JobSubmission`'s methods. Because all of the classes in the Auger job package provide data storage and organization, we found that the object coherence pattern was useful for all of them.

In conclusion we note that this design pattern emerged due to the application of our assertion creation process. Since the assertions we were deriving from the specification were far more sophisticated than typical programmer assertions, and because object-oriented interfaces usually encourage part-

```
public class JobSubmission {
    private List m_jobs;
    private String script;
    private boolean isCoherent;

    public JobSubmission()
    {
        isCoherent=false;
        ...
    }
    ...
    public void makeCoherent()
    {
        isCoherent=true;
    }
    /** invariant forall i: {0 .. m_jobs.size()-1}
        # ((Job)m_jobs.get(i) instanceof WorkJob ||
          (Job)m_jobs.get(i) instanceof FileJob) &&
          (!isCoherent || script != null)
    **/
}
```

Fig. 2. Example of Class Invariants Using Object Coherence Pattern

by-part initialization of objects, the creation of this design pattern was necessary to support the implementation of these assertions. Interestingly, our experiences suggest that design guidelines for class implementations can remove the need to use the object coherence pattern. For example, with careful design of the methods, it is possible to guarantee that no method will leave the object in a non-coherent state, thereby

VII. EVALUATION

In this section we critique the case studies we conducted, and then evaluate the benefits and costs of using specification-based assertions to reveal faults.

A. Threats to Validity

Perhaps the most serious criticism of the case studies we have presented is the use of automated and manual fault injection to simulate real faults. In an ideal evaluation, we would have instead used a system in which numerous previously discovered faults could be selectively enabled. Such systems are difficult to find, given that the development of most systems involves a mixture of corrective, adaptive, and perfective maintenance.

Second, the cost of performing the case studies limited our ability to evaluate the approach on large systems. Instead, we chose to evaluate the approach on small portions of real systems. Determining whether specification-based assertions can serve as a viable alternative to a test oracle for large systems remains a worthy area of investigation. However, we believe the best research strategy is to first develop more cost-effective methods of creating specification-based assertions, thereby lowering the overall cost and enabling the evaluation of the approach for large systems.

B. Benefits

The two case studies indicate that specification-based assertions can reliably reveal program faults. This work shows the

potential of using “complete” assertions as test oracles. Our specification-based assertions were much more powerful than traditional programmer-created assertions. When we executed the tests using only the pre-existing programmer-created assertions, none of the faults were in the Auger system, and only three of the eight were detected in the TDoG system.

There are two significant risks with the approach: that the specification is incorrect, or that the assertions are implemented incorrectly. In the former case, we can apply traditional automated and manual validation techniques. In the latter case, we note that assertions are generally simpler than independent oracle implementations, since they check results instead of recomputing them. Overall, the specification-based approach provides a complete definition of correctness that is easier to validate than an alternative oracle implementation.

Specification-based assertions remove the need to verify the output of the software under test. For example, when we were testing Auger’s job management subsystem, we were testing a module of the overall system that did not produce output at all. To identify the faults in this module using traditional unit testing, we would have had to create suitable testing infrastructure for exposing the internal state of the module in order to verify that it was correct. Another benefit of using assertions to detect faults is that for systems composed of self-checking modules, the assertions will fail in close proximity to the fault. This eases the task of locating the fault, and helps to detect faults whose effects are not propagated to the output.

We also believe that specification-based assertions offer benefits in terms of software design. We have already described the benefits of creating classes that avoid non-coherent object states. Employing this approach also encourages the encapsulation of assertion code with the functionality it checks. For example, all specification-based assertions for Auger’s `Job` class were placed within that class. The result is a piece of code that is self-contained and more likely to be easily reused. (Indeed, for design by contract languages, the entire specification is included with the module.)

C. Costs

The goal of this paper is to establish the basic feasibility of the approach, and to begin to identify opportunities for reducing its cost. In this section we discuss the sources of cost, and propose potential solutions.

First, the expressiveness of the assertion language can significantly affect the cost of implementing assertions. This was illustrated by the need for the `isPermutation()` function in Section III. Languages that do not support features such as the *items* construct make it more difficult for programmers to implement assertions, and increase the chance that they are implemented incorrectly.

Second, there are the semantic gaps that exist between formal specification and program code. As discussed in Section VI, there are certain issues that arise in program code that simply don’t exist in the realm of formal specifications, and vice versa. To some extent programming languages already provide some support for run-time verification of potential

faults such as division by zero. However, other semantics gaps such as the lack of null in specification languages are not as easily bridged. One approach is to design programming languages and specification languages together to avoid such problems. Another is to enhance the specification capabilities of design by contract languages, raising them in expressiveness to the level of traditional formal specification languages.

Third, formal specifications have a reputation of being difficult to develop. Our experience is that this reputation is somewhat overstated [42]—once an investment has been made in learning a formal method, useful specifications can be developed cost-effectively. Another way of viewing this cost is that it can be made incremental by using abstraction in the specification. The specification can be made more precise until the tester is satisfied. However, characterizing the nature of the tradeoff in fault detection effectiveness relative to the level of abstraction in the specification remains an open question.

Compared to the types of assertions usually written by programmers, this approach requires the user to add substantially more checking code to the system. Recall that we wrote 224 lines of assertion code for Auger’s job management package and 131 lines of assertion code for TDoG’s data recorder. In the original versions of the systems, programmer-created assertions accounted for only 13 and 29 lines of code, respectively. On the other hand, this cost combined with the cost of specification may still be less than the cost of manually computing expected outputs or developing a trustworthy oracle implementation. This is especially true if a large number of test cases are to be executed.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we have evaluated suitability of using rigorously developed assertions as a potential solution to the oracle problem. We performed two case studies on industrial systems in order to evaluate the feasibility of this approach and to better understand its costs and benefits. Our results indicate that the approach is an effective method of revealing faults that adversely affect the program’s state. We have also provided valuable insights into the process of converting specifications to assertions, and have identified a number of sources of cost in the approach. It is not yet clear whether the approach we used to develop the assertions is less costly than other solutions to the oracle problem.

Future work includes further evaluation of the approach, using larger systems having real, well-documented and prepared faults. Assuming that specification-based assertions can be shown to be effective for all types of systems, the cost must still be reduced for the approach to be widely practical. Therefore a secondary area of future work is to develop and assess methods of reducing the cost, such as those described in the previous section.

Combining self-checking software with automatic input generation enables automated testing, without the need for complex model checkers or other difficult to apply techniques. Given the computation power of today’s computers, harnessing

it to improve the reliability of the software that we develop is a worthy area of future research.

ACKNOWLEDGMENTS

We wish to acknowledge Cigital, Inc. for providing the TDoG and JavaWrap systems and Jefferson Lab for providing Auger. In particular, we acknowledge J. Richard Mills and Viren Shah of Cigital, Inc. We also acknowledge Michael A. Haddox-Schatz of Jefferson Lab, who developed Auger and Graeme Smith, the creator of Object-Z.

REFERENCES

- [1] M.-C. Gaudel, "Testing can be formal, too," in *TAPSOFT '95: Theory and Practice of Software Development*, ser. Lecture Notes in Computer Science, P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds., vol. 915. Springer-Verlag, 1995, pp. 82–96.
- [2] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, Nov. 1982.
- [3] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking system rules using system-specific, programmer-written compiler extensions," in *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, 23–25 Oct. 2000.
- [4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 1–25, Feb. 2001.
- [5] D. J. Richardson, S. L. Aha, and T. O. O'Malley, "Specification-based test oracles for reactive systems," in *Proceedings of the 14th International Conference on Software Engineering*. Melbourne, Vic., Australia: ACM New York, NY, 11–15 May 1992, pp. 105–18.
- [6] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated testing of Java programs," in *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2001)*. San Diego, CA: IEEE, 26–29 Nov. 2001.
- [7] J. Voas and L. Kassab, "Using assertions to make untestable software more testable," *Software Quality Professional*, vol. 1, no. 4, Sept. 1999.
- [8] J. M. Voas, "Software testability measurement for assertion placement and fault localization," in *Proceedings of 2nd International Workshop on Automated and Algorithmic Debugging*, St. Malo, France, May 1995.
- [9] J. M. Voas and K. W. Miller, "Putting assertions in their place," in *Proceedings of the International Symposium on Software Reliability Engineering*. Monterey, CA: IEEE, 6–9 Nov. 1994.
- [10] B. Meyer and D. Mandrioli, *Advances in Object-Oriented Software Engineering*. New York, NY: Prentice Hall, 1992.
- [11] B. Meyer, "Applying Design by Contract," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.
- [12] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, Jan. 1995.
- [13] R. W. Floyd, "Assigning meanings to programs," in *Mathematical Aspects of Computer Science*, ser. Proceedings of Symposia in Applied Mathematics, J. T. Schwartz, Ed., vol. 19. New York, New York: American Mathematical Society, 5–7 Apr. 1967, pp. 19–32.
- [14] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Oct. 1999.
- [15] L. G. Stucki and G. L. Foshee, "New assertion concepts in self-metric software," in *Proceedings of the 1975 International Conference on Reliable Software*. IEEE, 1975, pp. 59–71.
- [16] M. Müller, R. Typke, and O. Hagner, "Two controlled experiments concerning the usefulness of assertions as a means for programming," in *Proceedings of the International Conference on Software Maintenance*. Montreal, Quebec, Canada: IEEE, 3–6 Oct. 2002, pp. 84–93.
- [17] M. Blum and S. Kannan, "Designing programs that check their work," *Journal of the ACM*, vol. 42, no. 1, pp. 269–91, Jan. 1995.
- [18] W. Goerigk, T. Gaul, and W. Zimmermann, *Tool Support for System Specification and Verification*, ser. Springer Series Advances in Computing Science. London, UK: Springer Verlag, 1999, ch. Programs without Proof? On Checker Based Program Verification, pp. 108–23.
- [19] H. Wasserman and M. Blum, "Software reliability via run-time result-checking," *Journal of the ACM*, vol. 44, no. 6, pp. 826–49, Nov. 1997.
- [20] M. Brörkens and M. Möller, "Dynamic event generation for runtime checking using the JDL," in *Runtime Verification*, ser. Electronic Notes in Theoretical Computer Science, K. Havelund and G. Rosu, Eds., vol. 70. Elsevier, 26 July 2002.
- [21] K. Havelund and G. Rosu, "Monitoring Java programs with Java PathExplorer," in *Runtime Verification*, ser. Electronic Notes in Theoretical Computer Science, K. Havelund and G. Rosu, Eds., vol. 55. Elsevier, 23 July 2001.
- [22] Eiffel Software, "Building bug-free O-O software: An introduction to Design by Contract." [Online]. Available: <http://archive.eiffel.com/doc/manuals/technology/contract/>
- [23] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, and J. Kiniry, "JML reference manual," Apr. 2003, department of Computer Science, Iowa State University. Available from <http://www.jmlspecs.org>.
- [24] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass – Java with assertions," in *Runtime Verification*, ser. Electronic Notes in Theoretical Computer Science, K. Havelund and G. Rosu, Eds., vol. 55. Elsevier, 23 July 2001.
- [25] R. Kramer, "iContract – the Java design by contract tool," in *Proceedings of TOOLS 26: Technology of Object-Oriented Languages and Systems*. Santa Barbara, CA: IEEE, 3–7 Aug. 1998, p. 295.
- [26] D. C. Luckham and F. W. von Henke, "Overview of Anna, a specification language for Ada," *IEEE Software*, vol. 2, no. 2, pp. 9–224, Mar. 1985.
- [27] "Digital Mars - Design By Contract." [Online]. Available: <http://www.digitalmars.com/ctg/designbycontract.html>
- [28] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated testing based on Java predicates," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*. Rome, Italy: IEEE, 22–24 July 2002.
- [29] G. Smith, *The Object Z Specification Language*. Kluwer Academic Publishers, 1999.
- [30] J. A. Clark and D. K. Pradhan, "Fault injection: A method for validating computer-system dependability," *Computer*, vol. 28, no. 6, pp. 47–56, June 1995.
- [31] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting how badly "good" software can behave," *IEEE Software*, vol. 14, no. 4, pp. 73–83, July / Aug. 1997.
- [32] D. M. Blough and T. Torii, "Fault-injection-based testing of fault-tolerant algorithms in message-passing parallel computers," in *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*. Washington - Brussels - Tokyo: IEEE, June 1997, pp. 258–269.
- [33] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Proceedings of The International Conference on Dependable Systems and Networks*. New York, New York: IEEE, June 2000, pp. 417–426.
- [34] R. A. DeMillo and A. J. Offutt, "Experimental results from an automatic test case generator," *ACM Transactions on Software Engineering and Methodology*, vol. 2, no. 2, pp. 109–127, Apr. 1993. [Online]. Available: <http://www.acm.org/pubs/articles/journals/tosem/1993-2-2/p109-demillo/p%109-demillo.pdf>
- [35] M. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *IEEE Computer*, vol. 30, no. 4, pp. 75–82, 1997.
- [36] J. Voas and G. McGraw, *Software Fault Injection*. John Wiley and Sons, Inc., 1998.
- [37] J. M. Haddox, G. M. Kapfhammer, and C. C. Michael, "An approach for understanding and testing third party software components," in *Annual Reliability and Maintainability Symposium 2002 Proceedings*, Seattle, WA, 28–31 Jan. 2002, pp. 293–9.
- [38] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed. Hertfordshire, UK: Prentice Hall International Series in Computer Science, 1992.
- [39] M. Haddox-Schatz, "Auger design documentation," Jefferson Labs, Tech. Rep., May 2003.
- [40] J. Haddox, G. Kapfhammer, and J. Steven, "TDoG design documentation," Cigital, Inc., Tech. Rep., 1998.
- [41] "JUnit home page." [Online]. Available: <http://junit.org>
- [42] D. Coppit and K. J. Sullivan, "Sound methods and effective tools for engineering modeling and analysis," in *Proceedings of the 25th International Conference on Software Engineering*. Portland, Oregon: IEEE, 3–10 May 2003, pp. 198–207.