

An Evaluation of Specification-Derived Assertions

Matthew F. Curtis-Maury
Dept. of Computer Science
The College of William & Mary
Williamsburg, VA 23185
mfcurt@cs.wm.edu

Jennifer M. Haddox-Schatz
Dept. of Computer Science
The College of William & Mary
Williamsburg, VA 23185
jhaddox@cs.wm.edu

Abstract—Assertions are a mechanism that can be used to enhance the effectiveness of software testing. Where software testing is concerned with verification of a program’s output, assertions provide a method for verifying the internal values of a program. When used properly, assertions can provide assurance that a program is correct. However, as far as we are aware, there has been little research into methods for assertion creation. Too often, assertions written by programmers are often added to programs in a haphazard manner. Such assertions provide poor verification of system behavior because they only address low-level details of the implementation that have little relationship to each other or the overall high-level specification

We introduce a technique for assertion creation based on deriving assertions from a program’s formal specification. In order to evaluate our approach, we applied it to Nova-Solver, an existing fault tree analysis tool. We present preliminary results of comparing specification-derived assertions to traditional programmer-created assertions and discuss differences in the errors found by the two assertion creation approaches. From this experiment we report on the experience of manually translating a formal specification into assertions and the insights we gained into the strengths, weaknesses, and cost-effectiveness of specification-derived assertions.

I. INTRODUCTION

According to [1], formal methods and software testing are two general approaches used to verify software. Formal methods involve utilizing a mathematical process to show that the behavior of a system matches the function outlined in its original specification. Unfortunately, actually applying formal methods in practice is time consuming and expensive; thus, formal methods are used infrequently [1], [2].

The more commonly used approach to program verification is software testing. The goal of software testing is to identify faults in the software during development to help ensure that the software will not fail after deployment [3], [4]. Unfortunately, software testing does not perfectly address all of the difficulties of software verification. One problem is that exhaustive testing is an impossibility. Thus, we must settle for testing with some subset of the input space in order to determine if a program contains a fault. Further, even if the output from running these tests appears correct, we can not be certain that the program is error-free [2].

A mechanism that has been suggested and used as an aid to software testing is that of the *software assertion*. An assertion is defined as a self-check on a portion of a running program’s data state [5]. Where software testing is primarily concerned with verifying the output of a program, assertions go one

step further by verifying the internal values of a program. Assertions therefore offer a way to more quickly identify when a program has entered an erroneous state. Without assertions, the alternative is to wait for the program to complete execution and depend on the output produced to determine whether or not the program behaved correctly. Further, depending solely on output for verifying a program may allow some program faults to remain undiscovered. It is entirely possible that a program enters an anomalous state during its execution due to a fault, but the final output still appears correct [4], [6]. By using assertions, it may be possible to detect program faults that might otherwise go unnoticed. Thus, employing assertions can strengthen the process of software testing.

Despite the seemingly widespread support for assertions, resources detailing how to use assertions in an effective manner are limited. In fact, developers are unsure of how to write effective assertions [7], 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 which have little relationship to each other or the overall high-level specification

Additionally, assertions are not considered until the implementation stage of the software life cycle, and so it is the developers of the software who author the assertions. If a programmer introduces a bug into a piece of software, it is likely that the logical error that resulted in the bug will be reflected in the assertions written by that same programmer [6]. The aforementioned factors have the combined effect of reducing the effectiveness of the assertions that are used in a program and subsequently weakening the benefits that assertions can provide.

In this paper we introduce an approach for enhancing software testing that is based on utilizing a program’s formal specification to derive assertions. A formal specification is meant to give a declarative, precise, unambiguous and complete description of a program’s intended behavior. Formal specifications focus on the intended behavior of a software system, and so will provide a systematic approach of deriving assertions. A formal specification tells *what* a program is supposed to do, and the implementation details *how* the program is supposed to do it. Similarly, assertions embedded in a program are also meant to state *what* the program is supposed to do. Thus, a formal specification provides a logical basis on

which to build an approach for creating effective assertions. By utilizing a formal specification, we can create assertions capable of verifying the correctness of the overall system specification, as opposed to traditional assertions that simply describe some aspect of the software's intended functionality at a low level of abstraction [7].

In order to provide an initial evaluation of our approach, we applied it to Nova-Solver, an existing fault tree analysis tool, and its formal specification. Nova Solver, written in C++, is used for computing the reliability of fault tolerant computer systems [8]. The associated specification formalizes the dynamic fault tree approach to analyzing system reliability. We present preliminary results of comparing specification-derived assertions to traditional programmer-created assertions and discuss differences in the errors found by the two assertion creation approaches. Based on this evaluation, we report on the experience of translating a formal specification directly into program assertions and describe the insights we gained into the strengths, weaknesses, and cost-effectiveness of specification-derived assertions.

The rest of the paper is structured as follows. Section II provides a summary of prior research involving assertions. In Section III we present our new approach for creating assertions using a simple example program and its specification. In Section IV we discuss our plan for an initial evaluation of using a formal specification to derive assertions. Section V discusses the results of our experiment. Section VI provides an analysis of our results and a critique of the experiment itself. Section VII presents ideas for future work. Conclusions are given in Section VIII.

II. BACKGROUND

In this section we detail some previous work involving assertions. This research includes approaches for determining effective locations for assertions as well as categorizing assertions. However, we have not discovered a sound methodology for creating effective assertions in the literature.

A. Providing Support for Writing Assertions

Rosenblum addresses the fact that although assertions have been recognized as an important construct for detecting errors in software, assertions have seen little widespread use in practice [7]. Rosenblum attributes developers' reluctance to use assertions to a lack of usable assertion processing tools and a lack of understanding as to the types of assertions most likely to reveal program faults. In response to these two issues, a tool called the Annotation Preprocessor (APP) was built. APP was used to build several software systems, and the experiences with APP were documented. This documentation includes classifying assertions into categories and reporting which categories best detect faults. Rosenblum also states that once the use of assertions is more widely accepted, a need will arise to fully integrate assertions in the software lifestyle. In fact, Rosenblum makes the suggestion that it could be useful to derive assertions from formal requirements and high-level

design specifications, but we are unaware of any subsequent research efforts in this direction.

B. Assertions for Testability Enhancement

Voas et al. address the issue of *where* to place assertions within a software system such that the assertions are likely to uncover faults that normal testing would not identify [4]–[6]. This research is directly related to the concept of software testability, the tendency for software faults to be revealed during testing [4]. The ultimate goal of this work is to devise a strategy for determining which portions of the code are least likely to reveal faults (i.e. have a low level of testability), and place assertions at those locations. Voas et al. suggest the use of a dynamic testability technique called *sensitivity analysis* to achieve this goal. Sensitivity analysis is a three-step process meant to assign a testability rating to each location in a program, where a location is defined as a statement. Based on a location's testability score, one can determine whether or not an assertion should be created for that location. Though Voas et al. address *where* to place assertions within a program, they do not address the problem of writing effective assertions [6]

C. Design By Contract

Meyer et al. introduce a methodology for producing correct and robust software that involves assertions called *Design by Contract* [9]–[12]. In this approach a contract is specified between two pieces of code that will have interaction. This contract ensures that the client (i.e. caller) can expect to have a specific result returned to it, and that the contractor (i.e. callee) is not expected to carry out a task outside the specified scope. That is, the contractor is only obligated to return a correct result to the client if the client sent legal information to the contractor. These contracts are specified by assertions embedded in the code. Meyer et al. group these assertions into three categories [11]:

- 1) Preconditions express constraints to which any call must adhere to be correct.
- 2) Postconditions express properties that must be guaranteed upon return from that call
- 3) Invariants express properties that bind the state of a class. An invariant assertion must be satisfied after a class has been instantiated and over all invocations of the class's operations.

With this work, Meyer et al. provide a methodology for creating assertions that begins to shift the practice of writing assertions from the implementation phase to the design phase. That is, to employ Design by Contract, one must at least begin to consider how the interfaces in a software system will interact, which is more of a design issue than an implementation issue. However, the Design by Contract methodology does not provide a clear approach for exactly how to integrate formal specifications into the process of deriving assertions.

D. Other Work

Two other relevant pieces of work involve performing assertion-like checking not within the code but on an abstraction of the implementation. The TestEra framework for the automatic generation of test data and correctness criteria for Java programs is presented in [13]. In this work, a first order relational language called Alloy is used to provide a specification for program inputs, and then the Alloy Analyzer tool is used to generate test inputs meeting that specification. These inputs are translated into Java test cases, and the program under test is run. The resulting output is then translated back into Alloy in order to allow the Alloy Analyzer to verify the input and output against a correctness criteria, which is also specified in Alloy [13]. The authors of [14] introduce a framework for performing interface violation detection in component-based software systems. This research idea allows validation checks to be performed on both the input to and output from a component while keeping the code which performs the checking separate from both the component and any client that would invoke that component. This separation is accomplished by translating the actual component into an abstract representation by using mathematical models (i.e. sets, strings, functions, etc.) [14]. Upon invoking the component, the internal state of the component is transformed into an abstract model, and its preconditions are checked. Then the component is transformed back into its original representation so that the operation invoked can execute. Once this operation has finished execution, the internal state is converted back into a model such that postconditions can be verified. The component is then translated back into its original state and control is returned to the client that originally invoked the component in question. Though the approaches given in [14] and [13] introduce innovative ways for verifying program correctness, they do not address the problem of how to write a meaningful correctness criteria or effective precondition/postconditions.

III. USING FORMAL SPECIFICATIONS TO DERIVE ASSERTIONS

To clarify our approach for assertion creation, we illustrate its use on a simple example. Consider the following implementation of insertion sort, which sorts an array of Elements in increasing order and contains one simple assertion of the type a programmer might write involving the size of the array:

```
public void sort(Element a[], int size)
{
    Element save;
    assert(a!=null);
    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;
    }
}
```

Note that the assertion on its own is not powerful enough to guarantee the correctness of the sort algorithm, yet this is often the type of assertion that a programmer would write. Thus, we want to create stronger assertions that can guarantee the correctness of the algorithm.

To create stronger assertions for insertion sort, we look to a formal specification of sort in Z [15]:

[*Element*]

Element is a given type that represents an arbitrary item that can be stored in an array.

| $_{-} \leq _{-} : Element \leftrightarrow Element$

Here we define \leq for elements of the *Element* type.

$Sort : seqElement \rightarrow seqElement$ $\forall in, out : seqElement \bullet$ $Sort(in) = out \Leftrightarrow$ $items(in) = items(out) \wedge$ $(\forall i, j : 1..#out \mid i < j \bullet out(i) \leq out(j))$

Sort takes a sequence of elements as input and lists that same sequence of elements in increasing order as output. Note that we also rely on the *items* function in this specification for *Sort*. The *items* function ensures that the output sequence is a permutation of the input sequence.

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, the following informal description and the above formal specification should hold for *any* sorting algorithm whose goal is to sort elements in nondecreasing order:

- The elements of the list at the end of *sort*'s execution must be some permutation of the list's elements at the start of *sort*'s execution.
- For each pair of indices *i* and *j* in the list where $i < j$, $a[i] \leq a[j]$ must hold.

From the formal specification above we can derive several assertions. First, we have the assertion that the list of items must be finite, as sequences are finite by default. This is an assertion that obviously does not have to appear in the implementation, and hence represents an example of a disconnect between specification and implementation. Secondly, we can create an assertion corresponding to the predicate $items(in) = items(out)$. For this predicate we create a function called *isPermutation()*, and so we have the assertion:

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

For the predicate:

$\forall i, j : dom\ out \mid i < j \bullet out(i) \leq out(j)$

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. The implementation of `isPermutation` is omitted for brevity.

```

void sort(Element a[], int size)
{
    Element old_a[] = a;// make copy of list
    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]);
    }
}

```

Now that we have embedded the assertions derived from the formal specification, we know that as long as the assertions are not violated, the output produced, if any, **will** be correct, regardless of the input sequence or even the particular implementation of `sort`. Note though, that we can't guarantee that the program will terminate; for example, the assertions do not ensure that an infinite loop will not occur. If output is produced, the output will be an array of the elements in increasing order, and this ordering will be a permutation of the elements present in the array prior to calling `sort`. The same could not be said for the original `sort` implementation which contained a single assertion validating that the input array is not null.

IV. APPLYING THE NEW ASSERTION CREATION APPROACH

We now describe our preliminary experiment for gaining insight into the usefulness of our assertion creation approach. As our case study, we selected part of the Nova Solver tool, a real software system on the order of 10000 lines of C++, and its associated formal specification, written in Z.

A. An Overview of Nova Solver

Fault Trees are a means by which known rates of failure of independent subcomponents of a system can be used to predict the reliability of the system as a whole. The lowest level of a fault tree is the event, which may correspond to the failure of a subcomponent. Each event will have associated with it a distribution for its probability of occurring as well as a replication value. By representing the likelihood of system failure as the root of a tree with each node a logic gate and each leaf an event, one can propagate known event rates up

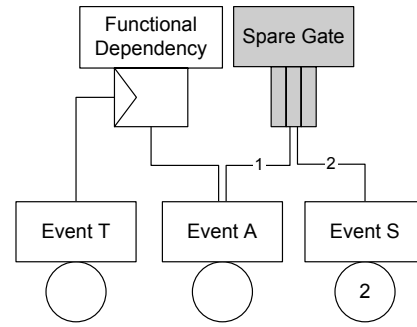


Fig. 1. dynamic fault tree example

the tree to the system level, which is itself an event, thereby determining its probability of failure. When the logic gates are simply AND OR, and threshold gates, a formula can be composed from a given fault tree to compute the system event rate. However, with the addition of priority AND gates, spare gates, functional dependencies, and sequence enforcers, as well as more complex trees, a different method of computing a system failure rate is necessary. An example of a fault tree appears in Figure 1. In the figure, Event T is the trigger event for the functional dependency; if T occurs, this causes Event A to occur. Events A and S are inputs to the spare gate and Event S has a replication value of 2, which indicates that the shape represents two components.

Nova Solver is a program which accomplishes exactly this task. Written in C++, Nova Solver can be viewed as a collection of modules which act as a pipeline beginning with the input of data and ending in the production of a system failure probability. This program takes as input three pieces of data. First among the input is a textual fault tree. As the name implies, this data is a fault tree stored in a text file. The second input is a basic event model (BEM) file. This file stores the failure model associated with each event to be considered by the program. The final input to Nova Solver is a list of parameters. These parameters include mission time and the definition of the system event. The mission time is the length of time for which system failure should be calculated. The system event is used to determine what constitutes a system failure, such that occurrence of the system event is equivalent to system failure.

Nova Solver begins by creating a `FaultTree` object from the textual fault tree provided as input to the program. This fault tree is represented by an instance of class `FaultTree` which is then passed to an instance of `FaultTreeSemantics`. `FaultTreeSemantics` is a class with member functions to act upon the given fault tree, as well as the system event input, and create from them an object of type `FailureAutomaton`. `FailureAutomaton` consists of a set of states and a set of transitions. This instance of `FailureAutomaton` is then passed to `FailureAutomatonSemantics` which uses this FA, in conjunction with the BEM passed from the BEM parser, to create an instance of a `MarkovChain`. This data is passed to `MarkovChainSemantics` which solves the Markov Chain

using a well-known algorithm. The program can compute the system reliability from this solved Markov Chain.

B. Description of Experiment

Since assertions are typically created by the programmer during development, we believe that comparing our specification-derived assertions to these programmer-created assertions provides a logical starting point for evaluating our assertion creation technique. The Nova Solver system already contained a small number of these programmer created assertions amounting to about 70 lines of code when it was given to us for use as a case study.

We created two copies of the Nova Solver system. One version was to contain programmer-created assertions, and the other was to contain specification-derived assertions. Different members of our research team took responsibility for managing each version. Because the original Nova Solver had a relatively small number of programmer derived assertions, the team member responsible for the version with programmer assertions had the task of adding additional assertions. In order to add correct and meaningful assertions, this team member had the task of becoming very familiar with the Nova Solver system works and how the different classes interact. He attained this understanding through meetings with the original developer, reading documentation, inspecting the Nova Solver code, and actually running the tool itself. Note though, that he was not given access to the Nova Solver’s formal specification. Access to the specification had the potential to bias the types of assertions he would write. Had this team member added programmer-created assertions that reflected the specification too closely, the benefits of this comparison would have been greatly reduced.

Several members of our research team took on the responsibility of translating the specification into assertions. This task required a solid understanding of the Z specification language and a very close familiarity with the Nova Solver specification itself. In order to ensure a proper understanding of the specification, the author of the Nova Solver specification periodically met with us to review the various sections of the specification. A careful understanding of the Nova Solver code was necessary to properly implement the new assertions, and this understanding was gained via code inspections and meetings with Nova Solver’s original developer.

Once all the assertions had been added to both versions of Nova Solver, the next step was testing. A test suite of both unit and system tests had already been created by Nova Solver’s developer; thus, we began our testing process with these test cases. We also developed additional unit and system tests to augment the existing test suite. Our goal was to compare the number of assertions failed in each version in order to determine whether or not one assertion creation method was more effective at identifying bugs or misuses of code than the other assertion creation method.

V. EXPERIMENTAL RESULTS

In this section we describe the outcome of our evaluation. We discuss the experience in terms of implementation cost for both our new approach and the traditional approach for assertion creation and report the results of running identical test suites on both versions of Nova Solver.

A. The Costs of Implementation

We first report on a comparison of implementing assertions using the traditional method and the method described in this work. The additional programmer-created assertions were implemented by one developer over a period of approximately 6 weeks. Once he gained an understanding of the code, the task was relatively straightforward as most of the assertions he added were simple. Examples of such assertions include asserting that some value is not null and asserting that a variable was equal to the correct value following an assignment statement. Our developer added approximately 150 lines of assertion code in addition to the assertions already placed in the system by the developer. In total, this version of the system contained approximately 220 lines of programmer-created assertion code.

The specification-derived assertions were implemented by 3 developers working over a period of 8 weeks for several hours per week. Even after a familiarity with the system and understanding of the code was gained, the development effort was non-trivial. Over 2300 lines of code were added to implement the specification-derived assertions. Many fairly complex functions had to be added to implement the specification. Thus, we were frequently in the situation of having to test, and in several cases debug, our assertions before they could be of any use.

From this experience, we conclude that creating specification-derived assertions adds an additional non-trivial development task to programming effort.

B. Results of Running the Existing Test Suite

The original test suite for Nova Solver consists of 30 system tests and 27 unit tests. All tests passed the version of Nova Solver that we were given for this evaluation. When we ran the test suite with the version of Nova Solver with programmer-created assertions, all test cases passed. When the test suite was run with the specification-derived assertion version of Nova Solver, two specification-derived assertions were violated by two of the system tests. Although these test case failures do not indicate major faults in the code, they both represent cases in which a class was misused due to deviation from the specification.

The specification states that a threshold value for a Threshold gate in a fault tree must be greater than 0. The code defines a class `Threshold` to represent a threshold value, and we created assertions to ensure that the value is greater than 0. The `Threshold` class is a subclass of class `Whole` which can take on values of zero. The default constructor for `Threshold` simply calls the constructor of the superclass, which sets a

value of 0 for the `Threshold` object. The system does utilize this default constructor, which resulted in the assertion failure.

Similarly, the specification defines a time value as being greater than zero. A `Time` class exists in the Nova Solver system to represent time values. This class is a subclass of class `Real` which can legally be assigned values of zero according to the specification. `Time` defines a constructor that takes one parameter. The constructor calls the constructor of `Real` that takes one parameter in order to assign the value to the object. We added an assertion to `Time`'s one argument constructor to ensure that it was never given a value of zero. However, this assertion was violated during testing, which leads us to the conclusion that `Time` objects are being constructed with illegal values.

The process of studying the system in order to write the specification derived assertions correctly resulted in two actual implementation errors being discovered by two members of the research team. The `FaultTree` class defined a copy constructor that did not copy the `system_event` field correctly. Though the process of implementing our specification-derived assertions is what allowed this error to be discovered, no assertion derived from the specification would have identified it. The second error was found in the one argument constructor of the class that encapsulates a probability. The `Probability` class is a subclass of class `Real`, and this constructor simply calls the one argument constructor of class `real`. The `Probability` constructor takes a parameter of type `long double`, but the `Real` constructor takes a parameter of type `double`. Thus, we have the potential for overflow.

C. Results of Running The Additional Test Suite

To supplement the results above, we created a new test suite consisting of 16 additional system tests and 68 additional unit tests. When we ran the test suite with the version with programmer-created assertions, only one test case, a unit test that will be discussed below, resulted in a violation of a programmer-created assertion. Running the new test suite with the specification-derived assertion version of Nova Solver resulted in 10 system tests failing due to assertion violations and 20 unit tests failing due to assertion violations. Note the specification-derived assertions were not violated due to major system errors, but were violated due to misuse of various components as well as illegal system inputs.

In addition to uncovering errors, the purpose of the system tests was to determine how the system handles invalid fault trees. Before going on, we note that the parser that translates the textual description of a fault tree into a `FaultTree` object contains checks that identified mistakes in the fault tree descriptions in both versions of Nova Solver. This checking code in the parser was actually based on the formal specification. Thus, in some cases the parser recognizes problems that would otherwise have been caught by our specification-derived assertions. However, for the system testing phase of our evaluation, we removed the code from the parser that performs some of these checks. The parser caused 9 of the system tests run with the programmer-created assertion version of Nova

Solver to fail. The other seven system tests passed. Note that the 6 system tests that did not fail due to specification-derived assertions failed because the parser identified the problem with the input first. We discuss the 10 test cases that resulted in violations of specification-derived assertions below.

In four of the test cases, the specification-derived assertions identified problems with the input that went unnoticed by both the parser and the programmer-created assertions. Below is a listing of the problem with each of these test cases:

- 1) An and gate was specified as an input to a spare gate
- 2) An or gate was specified as an input to a spare gate
- 3) A threshold of zero was specified for a `Threshold` gate
- 4) An input to a spare gate is specified as an input to another gate that is not a spare

The items in the list above all represent incorrectly defined fault trees, yet the programmer assertion version of Nova Solver did not detect the errors.

Additionally, we created three test cases in which fault trees with cycles were defined. With the cycle checking code removed from the parser, the version with programmer-created assertions produced output. The version with specification-derived assertions identified all cycles.

For the remaining three test cases, the problem with the test case was caught by the parser for the programmer-created assertion version, but was caught by assertions in the version with specification-derived assertions.

In order to test the correctness of each of the classes independently we created unit tests. These tests attempted to isolate subsections of the code to be tested. In so doing, we were able to create objects with exactly the characteristics we wished them to have. This provided us with the ability to see how well each component of the system handles objects without interference from the other components. The test results were overwhelmingly one sided.

Many of the unit tests constructed identical fault trees to those made in the system tests, the majority of which were invalid. All but one of the illegal objects created in the unit tests for `FaultTree` caused failed assertions in the formal-specification-derived assertion code. Additionally, we created test cases for three other classes (`FailureAutomatonTransition`, `FailureAutomatonSemantics`, and `FailureAutomatonState`) which produced invalid objects. These illegal objects were detected by the formal-specification-derived assertions.

In the programmer assertion code none of the illegal objects constructed in the unit violated any assertions. The programmer assertion code, however, did uncover the copy constructor bug described earlier. The bug was discovered in this code because the test case made use of the `FaultTree` copy constructor and the programmer assertion code contained a check to ensure that the copy had been accomplished successfully.

VI. EVALUATION

In this section we summarize the cost-effectiveness of our assertion creation approach and report on the benefits and limitations of specification-derived assertions based on our experimental results. We also provide a critique of our experiment.

A. Cost-Effectiveness

In this preliminary study of specification-derived assertions we can draw several conclusions regarding their strengths and weaknesses. First, although this approach to creating assertions results in extensive checking code being added to the system, actually implementing these assertions is a large task indeed. Where traditional assertions can be added casually as the programmer develops the code, specification-derived assertions can be a separate development project in and of themselves. Because of the high cost of implementing assertions using this approach, we believe this approach may not be appropriate for non-critical systems. For example, employing this approach for verification of a calendar program or even a word processor may not be the best use of resources. However, for software systems that are life-critical such as software embedded in a pacemaker or an airplane's control systems, devoting extra time and effort to implementing checking code based on the program's formal specification may be worthwhile.

B. Benefits and Limitations

A key limitation to the specification-derived assertion approach is the relatively high cost of applying it in practice, as mentioned in the previous section. Based on our experimental results, other limitations emerged.

Limitations of our approach became apparent with the discovery of the copy constructor and potential overflow errors introduced in Section V-B. Though the copy constructor bug was identified by code inspection and by one of the programmer-created assertions, it was completely ignored by the specification-derived assertions. This is due to the fact that it represents a very low-level error. A formal specification addresses high-level concepts and so may not be concerned with a low-level detail such as ensuring all fields are copied properly in a copy constructor.

The potential overflow issue is a subtle error that would typically not be identified by a traditional programmer-created assertion. However, such a bug would almost certainly not be identified with a specification-derived assertion either. The reason is that there are certain constraints and concepts that exist in a program that do not exist in a formal specification. Size constraints on variables are not normally represented within a formal specification; thus the problems of overflow and underflow do not exist. From this we see that specification-derived assertions are not particularly helpful for identifying such errors.

Finally, another important limitation of our approach is that it requires a specification for the system to which assertions will be added. Even if a specification exists, our approach requires that the code itself match the specification's structure,

at least to a certain extent. If this match does not exist, determining the appropriate locations to place the assertions could be very difficult.

Still we feel that the results from running both versions of the system with our new test suite indicate some potential benefits of specification-derived assertions.

We first consider the specification-derived assertion failures in the `Time` and `Threshold` classes described in Section V-B. In these cases, the deviation from the specification did not cause the program to behave incorrectly. However, such deviations have the potential to cause problems, particularly if one part of a program is consistent with the specification and another part with which it interacts is not.

A specification is meant to define high level rules that reflect interactions between various component's of a system. By deriving our assertions from the specification, we can gain some confidence whether or not the system is following these rules, as we saw with the assertion failures in the `Time` and `Threshold` classes. In contrast, programmer-created assertions typically address low level details that have little relationship to each other or the high level purpose of the system. Thus, the programmer assertions did not notice `Threshold` and `Time` objects were being constructed with illegal values.

Benefits of the specification-derived assertion approach also emerged from the experimentation performed with the additional test cases as described in Section V-C. First we consider the unit tests that constructed invalid objects, which resulted in assertions that check the state of those objects to fail. For these tests, assertions were the main form of verification which could occur during execution. For this reason it is not surprising that the formal-specification-derived assertions caused program failures more often, as many of these assertions were written specifically to ensure that constructed objects adhered to a legal format. In contrast, very few of the programmer assertions were written to identify illegal object construction.

For example, we created 18 unit tests for the `FaultTree` class, many of which constructed invalid fault trees. None of the problems with invalid fault trees were identified by the programmer-created assertions, but they all were identified by one of the specification-derived assertions. When the entire system is running, the parser is what constructs the fault trees. If we view these invalid fault trees constructed for the unit tests as simulations of errors that could have existed in the parser, then we can view the specification-derived assertions as being more helpful than the programmer-created assertions.

Assume we are running both versions of the system, and the parser contains an error or we reuse modules in a non-parser context that causes an invalid `FaultTree` object to be constructed. Running the version with programmer-created assertions would result in no assertion failures and an output value. One would now have to verify that the output was correct. This would require an oracle or that the output be verified manually. Even if it were determined that the output were incorrect, it may still be somewhat difficult to determine the source of the error since the program did terminate normally.

In contrast, these difficulties are reduced when relying on the specification-derived version of Nova Solver. In this case, the program would terminate during the construction of the `FaultTree`. There would be no output to check, and it would immediately be known that the problem was occurring in the `FaultTree` class. From this one could focus debugging efforts on the portions of the parser that invoke the `FaultTree` method in which the assertion failed.

We also believe that specification-derived assertions offer benefits in terms of software design. In the original Nova Solver, code for checking the validity of the a fault tree exists in the parser. Thus, the `FaultTree` class is dependent on this particular parser implementation to identify whether or not a particular `FaultTree` object is legal. Thus, the `FaultTree` class is coupled to this parser. The consequence of this is that if we wish to reuse the `FaultTree` class in a new version of Nova Solver or another system entirely, we are left with a class that performs no real validation checks on its input data or state. We must either reuse the parser along with the `FaultTree` class and potentially modify the parser to meet the needs of the new system, or disregard the parser and attempt to port the checking code from the old parser into the new system. Both of these options leave room for error. When the specification is used to guide the development of the checking code for fault trees, the result is a `FaultTree` class that is more independent and hence, more reusable. Deriving assertions for the `FaultTree` class from the specification encourages the placement of these assertions within the `FaultTree` class itself as class invariants. Therefore, the checking code is encapsulated with the functionality it is intended to verify making the `FaultTree` class a stand alone component.

C. Evaluation of Experiment

Though ideally our specification-derived assertions would have identified serious errors in Nova Solver, the fact that it did not is not surprising for two reasons:

- 1) We were working with a stable version of Nova Solver that had already undergone some testing and refinement prior to our working with it.
- 2) We were working with a relatively small test suite.
- 3) The code used for the case study was not very large

An ideal evaluation of our approach would involve using it on a real system as it is being developed. This would allow our specification-based assertions to be used during the initial testing of the system, which is when many program errors are likely to be discovered. Such a scenario would provide the opportunity to determine how our approach fares against existing testing techniques in identifying program errors. Since we did not have access to a real system that was in the process of being developed, our choices were to use an existing system or to design and build a toy system to do this evaluation.

We determined that using Nova Solver as our case study would provide a more beneficial assessment of the usefulness of this approach. Nova Solver is a non-trivial system that already had an associated specification. Thus, we believe that it provided a more realistic example of the difficulties and

challenges of translating a specification into code than a simple toy program and toy specification would have provided.

VII. FUTURE WORK

Because of the benefits gained from using specification-derived assertions as well as the limitations resulting from using Nova Solver as a case study in our initial experiment, we believe this topic deserves further study. Revealing program errors requires test inputs that cause those errors to manifest themselves via a failed assertion or incorrect output. For this initial evaluation, we were limited to developing additional test inputs manually and so had a very small test suite with which to work. An automated approach to test input generation may have resulted in the discovery of a larger number of errors in Nova Solver. In fact, several members of our research group have begun using the TestEra [13] framework to generate a large set of test inputs. Using this automated approach to test input generation in combination with specification-derived assertions, they have already discovered additional problems in Nova Solver that we did not uncover with our limited test suite. An extensive summary of these results will be reported in another paper.

In future work we hope to report in more detail the lessons learned in mapping a formal specification into assertions. In applying our approach to Nova Solver and several other existing systems, certain patterns emerged during the translation process. These patterns can be used to categorize different types of assertions that can be derived from a specification. Additionally, we have discovered types of errors that would not be identified via specification-derived assertions (e.g. assertions related to the maximum and minimum) through this work.

A true test of the effectiveness of our approach for assertion creation requires that we employ it as a system is being developed. In future research, we hope to identify a real system to which we can apply our methodology during the design and implementation stages. If our approach continues to show promise, the ultimate goal would be automating the process of translating a specification into assertions and then embedding them into the code. Automation would reduce implementation costs and perhaps make our approach more generally applicable.

VIII. CONCLUSIONS

Because software failure is still very much an issue in today's society, there exists a need to continue to improve the process of software development and testing such that software can be made more dependable. Because various researchers and developers recognize the potential usefulness of assertions, we believe that beginning to develop a methodology for assertion creation can move us toward improved development and testing practices.

In this work we have described our new approach for assertion creation and discussed an initial evaluation of this approach. We believe our evaluation highlights some of the potential benefits of using a formal specification to derive

assertions. We hope that the work outlined here will motivate future study of our assertion creation approach such that it can someday be applied in practice to strengthen and improve the process of software testing.

ACKNOWLEDGMENTS

The authors acknowledge Dr. David Coppit for allowing the Nova Solver tool to be used for our evaluation and assisting us in gaining a thorough understanding of his system and its formal specification. We also acknowledge Richard Dutton and Ashwin Mundra, members of our research group, who identified the copy constructor and potential overflow errors via code inspection detailed in Section V-B and assisted in the process of translating Nova Solver's formal specification into assertions.

REFERENCES

- [1] "Testability of object-oriented systems," Reliable Software Technologies, Tech. Rep. 95-01, Dec. 1994.
- [2] J. Voas and K. Miller, "Software testability: The new verification," *IEEE Software*, May 1995.
- [3] J. Voas, "A few assertions on information hiding," *IEEE Software*, Mar. 1997.
- [4] 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.
- [5] J. Voas and K. Miller, "Putting assertions in their place," in *Proceedings of the International Symposium on Software Reliability Engineering*, Monterey, CA, 6-9 Nov. 1994.
- [6] J. Voas and L. Kossab, "Using assertions to make untestable software more testable," *Software Quality Professional*, Mar. 1999.
- [7] D. Rosenblum, "Towards a method of programming with assertions," in *Proceedings of the 14th International Conference On Software Engineering*, Melbourne, Australia, 11-15 May 1992.
- [8] K. Sullivan and J. B. Dugan, "Galileo/assap: Dynamic fault tree analysis tool reference manual," Tech. Rep., Apr. 2002.
- [9] "Building bug-free o-o software: An introduction to design by contract," <http://archive.eiffel.com/doc/manuals/technology/contract/>.
- [10] "Design by contract: A missing link in the quest for quality software," <http://www.elj.com/eiffel/dbc/>.
- [11] B. Meyer and B. Mandrioli, *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1992.
- [12] B. Meyer, "Applying design by contract," *IEEE Computer*, vol. 25, no. 10, pp. 40-51, Oct. 1992.
- [13] 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*, San Diego, CA, Nov. 2001.
- [14] S. Edwards and B. Weide, "A framework for detecting interface violations in component based software," in *IEEE Computer Society Proceedings 5th International Conference on Software Reuse*, Victoria, Canada, June 1998.
- [15] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*. Prentice Hall, 1998.