A Study of the Effectiveness of Using Formal Specifications to Create Assertions

Matthew F. Curtis-Maury & Jennifer M. Haddox-Schatz

PROJECT SUMMARY

Though the software engineering industry is relatively new, software has become pervasive in modern society. Despite the fact that software is commonplace today, it often proves to be prone to error and generally undependable. When software does fail, the consequences can be as bad as from loss of revenue to loss of life.

Software testing is a common defense against such failures. The key goal of testing is to identify program faults and errors before the software is actually deployed. Unfortunately, software testing does not guarantee that a system will be free from program errors.

One mechanism that can be used to enhance the effectiveness of testing is that of the *software assertion*. An assertion is a self-check on a portion of a running program's data state. When used properly, assertions can provide runtime verification that a program is correct. However, because no sound method of assertion creation exists, assertions are not always used in a manner that maximizes their value during the testing process. Thus, misusing assertions can result in weakening the effectiveness of software testing.

We propose a technique for assertion creation based on deriving assertions from a program's formal specification. The goal of this work is to show that deriving a set of assertions from a program's formal specification and then embedding these assertions in the program is a worthwhile approach for identifying program errors during testing. In order to evaluate the effectiveness of specification-derived assertions, we will apply our approach for assertion creation to Nova Solver, fault tree analysis tool on the order of 10,000 lines of code, and its associated formal specification.

Intellectual Merit

The key issue addressed by this proposal is to develop a methodology for assertion creation and therefore improve the process of software testing. Though the benefit of using assertions to validate software has been touted by developers and researchers alike, in practice assertions are often used in a haphazard manner, if they are even used at all. In fact, to the best of our knowledge, no sound method for the creation of effective assertions exists in the literature. The result is that the assertions which are added to a program often address low-level details that have little relationship to each other or the high level purpose of the system as a whole. By deriving assertions from a program's specification, assertions can be embedded into the code that reflect relationships between the various components of a system. We intend to show that these specification-derived assertions can identify program faults more effectively than simple assertions created by the programmer during development.

Creating specification-based assertions necessitates a complete translation of the specification into program code. Thus, an additional issue we are addressing is the conceptual "gap" that exists between a program and its specification. We hope that the process which we employ to create these assertions will provide insight into how a specification maps to its implementation.

Broader Impact

Because software failure is still very much an issue in today's society, there exists a need to continue to improve the practice of software testing. The work proposed here seeks to make such an improvement by providing a process for the creation of effective assertions. Through the research proposed we hope to demonstrate that utilizing a program's specification to derive assertions is a fruitful approach for identifying program faults. Additionally, by applying our approach to a real system we hope to identify patterns and rules for translating a specification into assertions. Finally, because there are so few examples of how to write meaningful assertions in the literature, we hope that the work proposed here can begin to offer insight into this area and hence potentially advance the state of the art in the practice of software testing.

CAREER: Runtime Verification:

Integrating Formal Methods Into Common Software Development Practice

Matthew Curtis-Maury & Jennifer M. Haddox-Schatz

PROJECT DESCRIPTION

According to (10), formal methods and software testing are the 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 (10; 13).

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 (11; 14). Unfortunately, software testing does not perfectly address all of the difficulties of software verification. In fact, Dijkstra demonstrated that exhaustive testing is an impossibility. According to Dijkstra, if we consider a program which accepts two thirty-two bit integers as input and assume that the testing was performed non-stop at a rate of 1,000,000 successful test executions per second, then exhaustive testing of this program would be completed in about 585,000 years (10).

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 (12). 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 (14; 15). 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. As a result, developers are unsure of how to write effective assertions (8), 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 (15). The aforementioned factors have the combined effect of reducing the effectiveness of the assertions that are used in a program and subsequently weakening the process of software testing.

In this proposal we present an approach for assertion creation based on deriving assertions from a program's formal specification. Our goal is to show that the effectiveness of software testing can be strengthened with a sound methodology for assertion creation. We believe that using a program's formal specification as the basis for writing assertions, and then embedding these assertions into the implementation will prove to be a worthwhile approach for identifying program errors.

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 (8).

To evaluate this approach for assertion creation, we will apply it to a real software system, Nova Solver, and its formal specification. Nova Solver is a fault tree analysis tool written in C++ used for computing the reliability of fault tolerant computer systems (9). The associated specification formalizes the dynamic fault tree approach to analyzing system reliability. The core component of this evaluation will involve comparing the efficacy of specification derived assertions to assertions added by the developer.

1 Scientific Objectives

We have two key objectives that we hope to address by performing this research:

1. **Develop a methodology for assertion creation and demonstrate its effectiveness.** Though research has been performed that provides guidance as to where assertions should be used in a program (12; 14; 5; 6), we are not aware of any work that produced a sound approach for defining how to write effective assertions. Thus, the research we are proposing will begin to address the issue of what makes assertions effective, where assertion effectiveness is defined as the ability to identify program faults. To demonstrate the effectiveness of our approach, we intend to compare it to the traditional approach for assertion creation: assertions written by the developer during the implementation phase.

2. **Gain insight into the relationships that exist between a specification and the "self checking" code corresponding to it.** This objective, which is intricately tied to the first objective, will be fulfilled by applying our specification derived assertion approach to a real life system. Ideally, using a specification based assertion creation process would simply require the identification of correspondences between the specification itself and modules in the software. Clearly, however, there will be instances in which specification and code do not match. It is safe to say that not all predicates must be turned into assertions. For example, a predicate stating that a data structure is finite is already guaranteed in a concrete implementation. Similarly some predicates will be impossible to satisfy. An example of this is a data structure defined as infinite in a formal specification. There will also likely be instances in which a predicate should be turned into an assertion, but the decision of how the assertion should be implemented or where this assertion should be embedded into the code is not obvious. Thus, a question we hope to answer is how to best translate such parts of the specification into assertions and then embed them into the correct portions of the system.

To help clarify how we will meet these objectives, we next provide an illustration of our approach on a simple example taken from Ms. Haddox-Schatz's masters project proposal. 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)
{
  int 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 the insertion `sort` implementation above, we look to a formal specification of sort in Z (7):

$$[Element]$$

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.

$$Sort : \operatorname{seq} Element \rightarrow \operatorname{seq} Element$$

$$\begin{array}{l}\forall\, in, out : \operatorname{seq} Element \bullet \\ \quad Sort(in) = out \Leftrightarrow \\ \quad\quad items(in) = items(out) \wedge \\ \quad\quad (\forall\, i, j : 1 \mathinner{..} \#out \mid i < j \bullet out(i) \leq out(j)))\end{array}$$

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 returns a bag of elements for a sequence. Representing each sequence as a bag allows us to insure that the number of times a distinct element occurs in the input sequence is equal to the number of times a distinct element occurs in the output sequence (7). That is, we can be sure 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 insertions sort. In fact, the following informal description of the above 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 $i < j$, $a[i] <= a[j]$ 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:

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

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
  int 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. That is, 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.

2 Impact

Today's software is far from perfect. Problems at the design level, as well as the implementation level, result in a finished product that doesn't behave exactly as it is intended. A new tool is needed in the testing process to allow programmers to increase the correctness of their code.

Much research has gone into determining the value of assertions as a software engineering tool. At this point it is generally accepted that assertions, when used correctly, can be a powerful aid in the detection and removal of runtime errors (8). However, as with many other matters of good programming practice (e.g. the use of comments), they are not always used in a manner to maximize their value. There have been previous studies done to determine an efficient and effective means of assertion creation, for example Design-by-Contract (1; 5), but the problem remains. The authors propose a new method for creating assertions which may better utilize their power. If assertions are created from the formal specification for a program, then a more thorough check of runtime values of the program may occur, resulting in the production of a higher quality system.

One unfortunate effect of the difficulty in creating powerful assertions is that they may be left out entirely in the production of a system. Even if assertions are used they may be thrown in haphazardly, as an afterthought as mentioned above. The proposed method of assertion derivation provides a formula which may be followed by software developers. The very existence of such a formula may increase the likelihood that assertions are used at all. Further, use of this formula for assertion creation has the potential to reveal more software flaws during the testing process than traditional programmer created assertions, with the final result being an improvement in the overall quality of software. Additionally, because there are so few examples in the literature of how to write meaningful, effective assertions, we believe the work proposed here will offer insight into this area and ideally motivate future research in defining clear approaches for assertion creation. The result could be advancing the state of the art in the practice of software testing.

We also note that this research has the potential to change the way the software development community thinks about formal specifications by directly linking them to the software testing process. This research may provide insight on how to map a specification to its implementation as we apply our assertion creation approach to a real software system. Further, we hope that this work will make clear to software developers that taking the time to develop a formal specification for a system can provide practical and useful benefits to the important task of software testing. Additionally, the work proposed has the potential to motivate other researchers to experiment with combining formal approaches and software testing in order to make software more dependable.

3 Research Plan

This section describes the plan for executing this proposed research. In order to evaluate the notion of specification derived assertions, we will apply our approach to Nova Solver and its associated formal specification. Our evaluation will compare the results of performing testing with programmer-created assertions and with specification-derived assertions. In addition to deriving assertions from the existing dynamic fault tree specification and embedding them into the Nova Solver code, we will also add "programmer created" assertions to the system as the developer of Nova Solver did not employ assertions in the original implementation. Before we describe our plan to evaluate our approach for the creation of specification derived assertions, we briefly summarize the system that will be our case study.

3.1 Nova Solver

Fault Trees are a means by which to use known rates of failure of independent subcomponents of a system 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 its probability of occurring. 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 the tree to the system level, which is itself an event, thereby determining its probability of failure. When the logic gates are simply AND and OR gates and the tree is not very complicated a formula can be composed from a given fault tree to compute the system event rate. However, with the addition of threshold gates, PAND gates, spare gates, FDEPs, and SEQs, as well as more complex trees, a different method of computing a system failure rate is necessary.

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 rate. 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 BEM (basic event model) file. This file stores the rates 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 fault tree from the textual fault tree provided as input to the program. This fault tree is represented by an instance of class Fault Tree which is then passed to an instance of Fault Tree Semantics. Fault Tree Semantics 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 Failure Automaton. Failure Automaton consists of a set of states and a set of transitions. These data members represent the logical notion of a failure automaton. This instance of Failure Automaton is then passed to Failure Automaton Semantics which uses this FA, in conjunction with the BEM passed from the BEM parser, to create an instance of a Markov Chain. This data is passed to Markov Chain Semantics which solves the Markov Chain using a well known algorithm. From the solved Markov Chain, the program can compute the system reliability.

3.2 Evaluating the Specification-Based Assertion Creation Approach

We have already accomplished the first step of evaluating our assertion creation approach, which is acquiring a real system on which to apply the approach. Before we can begin writing specification derived or programmer created assertions for the Nova Solver, we must become intimately familiar with the details of the system and how it is intended to operate. We have already begun the process of familiarizing ourselves with the system by meeting with Nova Solver's developer. Additionally, we have obtained user documentation for the system as well as the dynamic fault tree specification.

Once a thorough understanding of Nova Solver and the dynamic fault tree specification has been acquired, we will begin the task of creating assertions for the code. Two versions of the Nova Solver system will be used: "programmer created" assertions will be used in one version of the system, and the specification derived assertions will be employed in the other. The member responsible for creating specification derived assertions will study the specification line by line, and attempt to translate the specification into assertions using the approach outlined in the sorting example in Section 1. Accomplishing this task will also require that this team member gain a solid understanding of Z, the formal language in which the dynamic fault tree specification is written. The member responsible for creating "programmer created" assertions will simply write assertions based on his understanding of the system. Note though that the member of the research team responsible for writing "programmer created" assertions for the system will not rely on the formal specification to gain an understanding of Nova Solver. The reason for this is to prevent this team member from being influenced by the specification when he is writing the assertions. Actually embedding the assertions into the code will either be done manually or with the aid of an assertion tool, such as Digital Mars.

After the “programmer-created” and specification derived assertions have been successfully embedded into the two versions of the system, the next task will be to run the two versions of Nova Solver on a common test suite. The developer of Nova Solver had created a test suite when he implemented the system and so we will reuse this test suite in our evaluation. If necessary, we will add additional test cases to the test suite. Finally, we will record and study results obtained from running both versions of each system on this common test suite. The goal is to determine which assertion approach reveals the most faults in the Nova Solver application.

Though the number of faults revealed will be the primary metric used in our evaluation, we will also consider the faults discovered via assertions in relation to the number of assertions used in each approach. That is, even if one approach to assertion creation revealed far more faults than the other, we must consider this in relation to the number of assertions that had to be created and embedded into the code to achieve this result. Thus, for each version of Nova Solver we will compute:

Number of faults discovered by assertions / Number of assertions in the code

This metric will allow us to consider the benefits gained by one approach to assertion creation with respect to the overhead incurred when using this approach.

4 Collaboration

In this section we discuss the assignment of tasks to research team members.

Mr. Curtis-Maury will be responsible for gaining a thorough understanding of the Nova Solver system by reading user documentation and studying the code. Based on his understanding of the code, he will insert “programmer created” assertions into one version of the Nova Solver tool. In order to acquire the background necessary to write such assertions, he will read on-line documentation and research papers to gain an understanding of how assertions are currently used. Note should be made that he will not utilize Nova Solver’s formal specification to gain an understanding of the system as that could influence the types of assertions he will write.

Ms. Haddox-Schatz will also be responsible for gaining a clear understanding of the Nova Solver system as well as its associated formal specification written in Z. She will have the task of translating the specification into assertions and then embedding them into a version of the code that is separate from the one that will be used by Mr. Curtis-Maury. Though Ms. Haddox-Schatz has some limited experience with Z and Object-Z, she will continue to utilize Z documentation to solidify her understanding of this specification language.

The original developer of the Nova Solver system has already created a large test suite for the system. This test suite will be used to test both versions of Nova Solver. Ms. Haddox-Schatz and Mr. Curtis-Maury will together analyze the results of running these test cases and report them.

Note should also be made of collaboration with another research group. Ashwin Mundra and Richard Dutton will be conducting a research project investigating the effectiveness of combining the specification based approach to creating assertions described here with an exhaustive testing strategy that deals with a limited portion of a program’s input space (i.e. testing with all inputs within a certain range). They will also be utilizing Nova Solver for their work. One member of their research team will focus on the exhaustive testing component of the project and the other on translating Nova Solver’s formal specification into assertions. Thus, this team member and Ms. Haddox-Schatz will actually be working together to translate the specification into assertions which can then be embedded into the Nova Solver code.

5 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.

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 (8). 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 research efforts in this direction.

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 (12; 14; 15). This research is directly related to the concept of software testability, the tendency for software faults to be revealed during testing (14). 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 (15)

Meyer et al. introduce a methodology for producing correct and robust software that involves assertions called *Design by Contract* (1; 2; 5; 6). 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 several categories (5):

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 (exclusive to object-oriented programming) 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 integrate formal specifications into the process of deriving assertions.

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 (4). 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 (4). The authors of (3) 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.) (3). 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 (3) and (4) 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.

6 Summary

In this proposal we have introduced an approach for the creation of effective assertions based on formal specifications. Though various researchers and developers acknowledge the usefulness of assertions for software testing, no sound process exists for the creation of meaningful, effective assertions. The goal of the work proposed is to define a methodology for assertion creation that will result in the discovery of more program faults than assertions haphazardly added to systems by the programmers. We have defined our approach as well as a method for evaluating it. Our evaluation will involve testing a real application with both programmer-based and specification-based assertions and analyzing the results. We hope that the sound approach for assertion creation outlined in this proposal can provide insight into how to write effective assertions in practice and hence strengthen the process of software testing.

7 Acknowledgments

The authors wish to acknowledge Dr. David Coppit for allowing us to use his Nova Solver tool as the case study for our evaluation, and for taking time to share with us information regarding the design of his system.

CAREER: A Study of the Effectiveness of Using Formal Specifications to Create Assertions

Matthew F. Curtis-Maury & Jennifer M. Haddox-Schatz

References

[1] Building bug-free O-O software: An Introduction to Design by Contract. http://archive.eiffel.com/doc/manuals/technology/contract/

[2] Design By Contract: A Missing Link In The Quest For Quality Software http://www.elj.com/eiffel/dbc/

[3] S. Edwards, B. Weide, J. Hollingsworth. A Framework for Detecting Interface Violations in Component-Based Software. *IEEE Computer Society Proceedings 5th International Conference on Software Reuse* Victoria, Canada (Jun. 1998).

[4] D. Marinov, S. Khurshid. TestEra: A Novel Framework for Automated Testing of Java Programs. *Proceedings of the 16th IEEE Conference on Automated Software Engineering*. (2001).

[5] B. Meyer. Design By Contract. D. Mandrioli, B. Meyer. *Advances in Object-Oriented Software Engineering*. Prentice Hall (1992).

[6] B. Meyer. Applying ”Design by Contract”, in Computer (IEEE), vol. 25, no. 10, October 1992, pages 40-51.

[7] B. Potter, J. Sinclair, D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall (1996).

[8] D. S. Rosenblum. Toward a Method of Programming with Assertions. *Proceedings of the 14th International Conference On Software Engineering*. (1992).

[9] K. Sullivan, J. Bechta Dugan. Galileo/ASSAP: Dynamic Fault Tree Analysis Tool Reference Manual. Version 3.0.1 (April 2002).

[10] Testability of Object-Oriented Systems. Technical Report 95-01, Reliable Software Technologies, (Dec. 31, 1994).

[11] J.Voas. A Few Assertions on Information Hiding. *IEEE Software* (Quality Time Column). (March 1997).

[12] J. Voas, K. Miller. Putting Assertions in Their Place. *Proceedings of the International Symposium on Software Reliability Engineering*. Monterey, CA. Nov. 6-9, (1994).

[13] J. Voas, K. Miller. Software Testability, The New Verification. *IEEE Software*. (May 1995).

[14] J. Voas. Software Testability Measurement for Assertion Placement and Fault Localization. *Proceedings of 2nd International Workshop on Automated and Algorithmic Debugging*. St. Malo, France. (May 1995).

[15] J. Voas and L. Kossab. Using Assertions to Make Untestable Software More Testable. *To appear in Software Quality Professional*