

# **Integration of Assertions with Bounded Exhaustive Testing**

Ashwin Mundra Richard Dutton

## **PROJECT SUMMARY**

In critical software systems whose malfunction would have severe consequences, traditional testing has been insufficient to prove its validity.[2]

In small software systems where the test data set is finite, it is at times possible to generate all possible test cases and exhaustive testing is practical. Unfortunately the functional domain is often infinite or sufficiently large, making exhaustive testing quite unfeasible or impossible. This is the case when traditional testing is normally used. In the case where the software system in question is of great importance to human life, traditional testing does not provide complete assurance that the system will function correctly every time. We propose bounded-exhaustive testing, a method of combining the aspects of traditional testing and exhaustive testing. In this way, we hope to find a form of testing that will bring complete quality assurance for software systems in important real-life situations.

We will use bounded-exhaustive testing on a well documented system, in order to check whether this method catches errors which were not detected by the traditional testing. The approach involves defining the formal specification of the system from its documentation. Based on the formal specification, corresponding assertions would be embedded in the source code. These assertions can be used to act as runtime checks to validate the output from the test data sets.

### **Intellectual Merit**

The main goal of this research is to develop a testing methodology which has the efficacy of exhaustive testing but is feasible in large software systems.

We do not yet know how to ensure the total reliability of software systems without completely testing every element of the test data set. We lack a method for generating every equivalence class of a given test data set in order to obtain the aforementioned total reliability of the system. It has been shown that there is no algorithm to find consistent, valid, and complete test criteria.[10] This confirms that complete testing is a very difficult process. We propose developing an instance to find a test set that is large enough to span the domain and small enough that testing can be realistically performed for each element of the test set in order to solve this problem.

We do not yet know how bounded exhaustive testing and formal specification based assertions can work in conjunction to provide a more reliable testing method. Although research has been done in bounded exhaustive testing at the unit level, we are unaware of a methodology for combining the use of the assertions with bounded testing at the system level. We hope this method will detect errors previously unidentified in a system, thereby showing the combination of testing and assertions to improve the quality of software systems.

### **Broader Impact**

This proposal seeks to address the problem of testing software at a reasonable cost and time. The research seeks to bolster the reliability of critical systems, whose failure can result in loss of time, revenue, or life. In addendum, bounded-exhaustive testing can have a significant impact on quality assurance of software systems, both in industry and in academia.

# Integration of Assertions with Bounded Exhaustive Testing

Ashwin Mundra Richard Dutton

## PROJECT DESCRIPTIONS

Testing is an important aspect in all stages of a software's life-cycle. The reason for software testing is to identify bugs in the development stages so that we do not run into problems once the software is deployed. Program testing by using the representative data and comparing the actual results with the expected results has been used traditionally to determine errors. However, traditional testing has been often time-consuming, difficult and inadequate. Moreover, it is not an adequate proof of correctness, since it does not guarantee that system will work as expected under untested inputs. [3]

On the other hand, exhaustive testing is unfeasible because it involves testing every element in the test domain. In fact, it has been shown that exhaustive testing is impossible for large systems.[10]

In this project we will combine formal methods and bounded exhaustive testing for verification of software systems.

Formal specification and assertions are two useful methods that have been used for effective testing. Formal specification implies a mathematically precise, unambiguous, and comprehensive description of the software. Such specification can be validated either by informal expert inspection or by formal tool-assisted theorem proving technique. A simple explanation of formal specification would be that it is what the software should do. Similarly, assertions embedded in a program also verify what the software should do. Assertions can also provide us with runtime checks. Thus deriving assertions based on formal specification seems logical to identify the run-time errors in the system. This would also buttress the reliability of the system.

We will use a well documented and fairly large system call Nova Solver to test our approach of integrating formal specifications and bounded exhaustive testing. The Nova Solver, which is a reliability computing system, takes a fault tree as its input and computes the reliability of it through a series of computations. Thus, we should be able to identify bugs which were not detected in the initial testing of the system.

## 1 Scientific Objectives

Some components of our research use methods and approaches that are already in use today. While not used in practice by all software engineers, the idea of creating formal specifications for software systems is not a new idea. The use of assertions as a means of ensuring runtime correctness is a common practice. However, assertions are often created during programming as opposed to being created from the formal specification.

The following example will show the manner in which we plan to combine assertions with bounded testing, illustrating the scientific objectives of our research.

```
list<> r = Sort(list<> r, r.length);
list<> Sort(list<> inList, int sizeOfArray) {
    for (int i = 0; i < sizeOfArray; i++) {
        int temp = inList[i];
        for (int j = i; j > 0 && (inList[j-1] > temp); j--)
            inList[j] = inList[j-1];
        inList[j] = temp;
    }
}
```

The following is a specification written for the above *Sort* algorithm. The specification is written in Z,[7] a popular software specification language.

$$\begin{array}{|l}
\text{Sort : seq } \mathbb{Z} \rightarrow \text{seq } \mathbb{Z} \\
\hline
\forall in, out : \text{seq } \mathbb{Z} \bullet \text{Sort}(in) = out \Leftrightarrow \\
\text{items}(in) = \text{items}(out) \wedge (\forall i, j : 1 \dots \#out \mid i < j \bullet out(i) \leq out(j))
\end{array}$$

The specification has two separate sections. Above the horizontal line states that the input will be a finite list of numbers and the output also will be a finite list of numbers. Below the line, the specification gives the requirements of the *Sort* algorithm. The two conditions are the requirements the system must satisfy in order to know that *in* was put through *Sort*, resulting in the output of the correct list *out*. The first condition states that *out* must be a permutation of *in*. The second condition states that for any two entries in the list at positions *i* and *j* in the output where  $i < j$ ,  $out[i]$  must be less than or equal to  $out[j]$ .

The specification can be converted into assertions which can be embedded into code. The following illustrates the implementation of the assertions into the *Sort* algorithm from above.

```

list<> Sort(list<> inList, int sizeOfArray) {
    int *oldList = copyArray(inList, sizeOfArray);

    for (int i = 0; i < sizeOfArray; i++) {
        int temp = inList[i];
        for (int j = i; j > 0 && (inList[j-1] > save); j--)
            inList[j] = inList[j-1];
        inList[j] = temp;
    }

    assert(IsPermutation(oldList, inList, sizeOfArray));
    for (int i = 0 ; i < size - 1 ; i++)
        assert( inList[i] <= inList[i+1] );
}

```

Upon execution of this function, we will know one of two cases occurred: the function ran and failed an assertion meaning the algorithm was not written correctly, or the function executed to completion without failing an assertion. In the latter case, we have proof of the correctness of the algorithm.

The next step, in terms of our research, would be doing bounded-exhaustive testing on the *Sort* algorithm above with the embedded assertions. We will investigate methods for enumerating the possible input. Using the *Sort* algorithm example, enumeration would involve looking into two possible areas of the input:

**1. Length of the list *r***

The range of possible lengths of the list *r* go from  $r.length = x, 0 \leq x < \infty$

**2. Characteristics of the elements of list *r***

The list could contain many different combinations of numbers. The number of distinct numbers in the list can range from 1 to the length of the list. The list could also contain certain types of values, i.e. only positives, only negatives, positives and negatives, zero, etc.

One way to enumerate the possible inputs of *Sort* would be to look at the length of the list in conjunction with the possible number of distinct values.

Length of <i>r</i>	Distinct values in <i>r</i>
0	0
1	1
2	1, 2
3	1, 2, 3

The enumeration could be extended to also take into account the permutations of a list of the same order with a different ordering of the elements.

In order to show that a combination of bounded-exhaustive testing and formal specification-based assertions is a strong method of testing software systems, we must overcome a number of obstacles:

1. **We do not yet know how to ensure the total reliability of software systems without completely testing every element of the test data set.**

Before testing for complete correctness is plausible, there must exist a method of testing which can functionally test the whole domain of input without exhaustively testing the whole input. A form of bounded exhaustive testing is needed to test large portions of the test data set. This type of testing must take into account the different forms and possibilities of input to the software system. The method must seek to selectively test only certain data. There must be a way to partition the test data sets into some type of equivalence classes in order to group the data which behaves similarly when given as input. Because of this, close attention must be paid to the scope of the input. We will also need to avoid data repetition that can result from cases such as symmetry of the test data.

2. **We lack a full understanding of a methodology for effectively creating assertions from a system's formal specification and embedding them into the source code.**

The assertions we create will be completely based upon the formal specification of a software system. While this is not a new idea, we are hoping to find effective ways for embedding these assertions into the code. It is possible that the assertions from the formal specification do not match exactly what is needed or possible in the code. We will have to do some converting of the formal specification-based assertions in order to match what is happening in the code.

3. **We do not yet understand the way bounded-exhaustive testing and formal specification-based assertions can work in conjunction to provide a more reliable testing method.**

The research is intended to demonstrate the ability of bounded testing along with assertions to show reliability of a system. The objective of these assertions is in validating the correctness of the system at runtime. This objective is key to the research because of the nature of testing we will be performing. Since bounded-exhaustive testing will entail the testing of a large quantity of inputs, we need the assertions to verify the correctness of the system for every input we test. By finding more bugs in this manner which were not identified by the initial test of the system, we hope to show the use of bounded-exhaustive testing and formal specification-based assertions to be a advantageous practice.

## 2 Impact

Formal specification-based assertions combined with bounded exhaustive testing hopes to provide a practical, low cost approach to enhance the quality of software. One of the biggest obstacles in software engineering is the ability to test software at a reasonable cost and time. The proposed method of testing promises to target this problem.

Assertions have always been an important way of conducting runtime checks[6]. Formal specification based assertions provide runtime verification of the system. This approach has the potential to change the way software engineers develop test cases, thus we can have higher confidence than with traditional testing. By augmenting traditional testing with bounded exhaustive testing, we can bolster the reliability of critical software systems.

In developing critical software systems, undetected bugs often lead to incorrect reliability estimates. [1] This might lead to mammoth problems and artificially high reliability estimates. Our methodology would generate better reliability estimates with higher confidence.

During the testing phase, there is a need to compute the expected output for any given input to the system. Through the use of formal specification based assertions, we can guarantee the correctness of all output from the system. This allows for two key improvements to testing. The first is the fact that computation of expected outputs is no longer necessary. This can save time and money. The second improvement is that the proposed method of testing includes bounded exhaustive testing, meaning more of the possible inputs to the system will be tested. It is our hope that this combined effort of assertions with bounded exhaustive testing will therefore have an impact on the quality assurance of software systems.

### **3 Research Plan**

This section will describe our plan for doing the proposed research. The research is focused on the use of bounded-exhaustive testing in conjunction with the use of assertions based on formal specification to find bugs in programs.

#### **3.1 The Case Studies**

We will be utilizing Nova Solver, a reliability-testing software system to assess the efficiency of our testing method. The Nova Solver was developed by David Coppit at the University of Virginia during his Ph.D. research. The Nova Solver takes a fault tree as its input, and computes a probability of failure for this fault tree through a number of conversions and calculations. While the exact structure and inner workings of the Nova Solver are not important to the research goals of the proposed work, it is important to note that the system has a well-defined formal specification which will be used for deriving formal assertions. Because both members of the research group are new to the Nova Solver software, the preliminary stages of research have been and will continue to be temporarily focused on learning the intricacies of the system pertinent to the research goals, such as the domain of the possible inputs. The main goals of the research will be to find bugs that have thus far gone undetected after traditional testing and academic use of the software for a few years. Bugs have been detected through the use of the software, but we hope that our testing method will reveal additional errors in the code.

#### **3.2 Approach**

In the research, there will be two distinct focuses. The first will be in implementing assertions derived from the formal specification. The second will be in implementing a modified version of exhaustive testing, which will be referred to as bounded exhaustive testing. The final goal of the research is in allowing these two focuses to work together to test the system. In order to implement assertions, we must have knowledge of a few different entities. The first is the formal specification of the Nova Solver, which is written in Z. [7] We will need to quickly become familiar with both the syntax and semantics of Z in order to be able to read and understand the formal specification. Once we are familiar with the formal specification, we will then begin to develop the more basic assertions to become more comfortable with the art of formal specification-based assertions. Up to this point, it will not be necessary to be familiar with the actual coding of the system. However, before we begin inserting these assertions into the code, we will have to do a conversion from what the assertion would be solely based upon the formal specification to an assertion that can actually be implemented in the code. The hope is that there will be little to no change in the meaning and power of the assertion after it is converted. The second focus of the research is the idea of bounded exhaustive testing. The plan is to test substantial portions of the functional domain without testing all possible inputs. For this section, we will need to be familiar with the coding of the Nova Solver, which is written in C++. We have previously reviewed and run the system with David Coppit, the developer of the software. However, as it was an introduction to the Nova Solver, this was a high level review of the system. The input possibilities and

formats will be of extreme importance for our bounded exhaustive testing. Many of the possible inputs have been previously generated and tested by Coppit. Our goal is to find a larger quantity of test cases as well as finding more powerful test cases which give insight into errors in the software. The bounded exhaustive testing and assertions will then be used in combination.

### **3.3 Evaluation Criteria**

The purpose of conducting the research is manifold. The first goal is to understand the pros and cons associated with the approach of integrating formal specification derived assertions with bounded exhaustive testing. Secondly, we will be able to use the knowledge gathered about the obstacles to assist in the planning of the remaining details of the research effort which could be used as Ashwin Mundra's 710 research project.

In terms of errors the case study would give us insights about distribution of errors. A key to know would be where most of the bugs or errors happen in terms of the scope of the input.

The case study will also help us decipher more information about the bounds. For example, the functional domain is often infinite or sufficiently large. But the occurrence of events becomes more and more rare as we start going away from the base case. At one point the probability of that event occurring might become so infinitesimal that we might be able to ignore it. Similarly, our bound for testing will be dependent on occurrence and complexity of certain events or situations. We hope to be able to derive better bounds for exhaustively testing the system.

## **4 Background**

In this section we will detail some previous research done on assertions and exhaustive testing.

### **4.1 Assertions**

The use of assertions in programs to induce runtime checks is not uncommon. There has been a significant research done on assertions. Also there are a myriad of research projects which have tried to enhance the power of simple assertions. Meyer's design-by-contract [5] is one of the most famous ones. Meyer provides a different approach for inserting assertions in object-oriented programs. He specifies preconditions and postconditions which check the interaction between two modules or classes. But he does not proceed to evaluate assertions based on formal specifications.

The paper by Voas and Miller [8] describes a middle path between no assertions and assertions at every location. Therefore, assertions based on formal specifications is different from their approach because we are mathematically trying to prove the correctness of the system.

TestEra by Khurshid and Marinov[4] is a novel framework for testing of java programs where they run the method on each test input, and use the method postcondition as a test oracle to check the correctness of each input. On the other hand, we use formal specification based assertions as our oracle.

### **4.2 Testing**

Exhaustive testing has been proved to be infeasible and impractical. Usually, the test data set is infinite or significantly large, which creates a situation that testing each element in the test domain is not pragmatic. For this reason there has been research conducted in choosing representative elements from the functional domain to make testing practical.[10]

TestEra deals with this issue by generating a bounded test data set from the precondition, on being given a formal specification for a method. In their future research, TestEra's authors plan to extend the capability

of TestEra so that users can decide the point which they think that enough test has been conducted. We will use bounded exhaustive testing instead of structural code coverage discussed in TestEra.

Other research have also shown that high reliability of a system is proportional to the amount of testing. Butler and Finelli in their research state that reliability growth models are portrayed to be incapable of overcoming the need for exorbitant amount of testing.[2]

Pseudo-exhaustive testing has also been conducted in the hardware world specially in VLSI chips. The paper by Jone, Rao and Chang describes the effective generation of pseudo-exhaustive test patterns for combinational VLSI circuits.[9]

Since pseudo-exhaustive testing has be done at hardware level and limited exhaustive testing has been done at the unit level in software, we propose bounded exhaustive testing to system level.

## **5 Collaboration**

We would be collaborating with one other group (Mathew F. Curtis-Maury and Jennifer M. Haddox-Schatz) in inducing the formal specification based assertions. Their research focuses on doing a comparison between programmer inserted assertions and formal specification derived insertions. We would be working with one member of their group to insert formal specification based assertions in the program code.

## **6 Summary**

Formal Specification based assertions and bounded exhaustive test is a promising approach to buttress the reliability of critical systems. This will help in saving both time and cost in software testing. Although some work has been done in assertions and pseudo-exhaustive testing, combining both of them is a novel concept. While the research will have a positive effect in the quality assurance of software systems, we are far from being able to generate an optimum bound for exhaustive testing. We propose to develop new techniques to help reach the goal of enhancing reliability of life critical software and simultaneously understand the theoretical shortcomings of the correctness guarantees provided by the method.

## **7 Acknowledgement**

We would like to acknowledge Dr. Coppit for his contribution to our research proposal. We also want to extend our gratitude to him for letting us use his Nova Solver as our case study.

## Integration of Assertions with Bounded Exhaustive Testing

Ashwin Mundra Richard Dutton

### References Cited

- [1] Paul E. Ammann, Susan S. Brilliant, and John C. Knight. The effect of imperfect error detection on reliability assessment via life testing. *IEEE Transactions on Software Engineering*, 20(2):142–8, February 1994.
- [2] Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.
- [3] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972.
- [4] Darko Marinov and Sarfraz 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, 26–29 November 2001. IEEE.
- [5] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
- [6] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.
- [7] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
- [8] Jeffrey M. Voas and Keith W. Miller. Putting assertions in their place. In *Proceedings of the International Symposium on Software Reliability Engineering*, Monterey, CA, 6–9 November 1994. IEEE.
- [9] S. C. Chang W. B. Jone, J. C. Rau. A tree-structured lfsr synthesis scheme for pseudo-exhaustive testing of vlsi circuits. *IEEE*, October 1998.
- [10] Martha A. Branstad W. Richard Adrion and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM*, 14(2), June 1982.