Integration of Formal Assertions with Bounded Exhaustive Testing for Critical Systems

Richard Dutton
Dept. of Computer Science
The College of William & Mary
Williamsburg, VA 23185
radutt@cs.wm.edu

Ashwin Mundra
Dept. of Computer Science
The College of William and Mary
Williamsburg, VA 23185
mundra@cs.wm.edu

***Abstract*— It is well-known that testing is only a partial proof of a program's correctness. One way to bolster testing is to add assertions as runtime checks to validate the execution of a system. Combining assertions with testing is compelling because the tester need not determine the correct output for a given input to the system–the assertions provide a self-check. However, this approach has two key problems. First, it is heavily dependent on the input data used during execution. Second, it depends on the completeness and quality of the assertions. In this paper, we present a testing approach in which assertions are derived from a formal specification, and the resulting self-checking program is subjected to a bounded exhaustive set of test inputs. Because they are derived from a specification, the assertions are more substantial. Exhaustively testing a subset of the inputs reveals errors which may be overlooked by traditional input partitioning schemes. To evaluate our approach, we applied it to a well-specified system, and assessed the ability of the approach to identify errors which were not detected by the previous application of a traditional unit and system testing approach. We found that we were able to find several new errors using our approach. However, the creation of assertions from specifications was a time-consuming process, as was the execution of test cases.**

I. Introduction

Testing is an important aspect in all stages of a software's life-cycle. The reason for software testing is to identify bugs during development so that the software is reliable once it has been 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 often been 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. [1]

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. [2]

In this project we have combined formal methods and bounded exhaustive testing for verification of software systems. From the formal specification for a system, we translate the constraints of the system into assertions. These assertions served as runtime checks to validate the output of the system. Next we test all possible inputs of the system up to a given small bound. If any of the runtime checks failed, we have a concrete example of an input which causes the system not to meet the requirements of the formal specification. If the system executes normally, we have guaranteed the validity of the system for this input.

Formal specification and assertions are two useful methods that have been used for effective testing. Formal specification gives a mathematically precise, unambiguous, and comprehensive description of the software. Such specifications are often validated by informal expert inspection or by formal tool-assisted theorem proving techniques. 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 additionally provide us with runtime checks. [3] 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. This approach has the potential to change the way software engineers develop test cases, thus we can have higher confidence than with traditional testing.

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 output from the system. Therefore the computation of expected outputs is no longer necessary.

In developing critical software systems, undetected bugs often lead to incorrect reliability estimates. [4] This might lead to problems and artificially high reliability estimates. Our methodology would generate better reliability estimates with higher confidence.

We used a well-defined system called Nova Solver to test our approach of integrating formal assertions 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. We added assertions based on the formal specification for the system. After we had added the assertions, we performed bounded exhaustive testing. We generated all possible inputs up to a certain small bound, then tested the system for each of these inputs. From this we were able to identify bugs which were not detected in the initial testing of the system. Through this combination of formal assertions with bounded testing, we also discovered flaws in the system's test suite in addition to

instances where the system did not match its specification.

The rest of the paper is structured as follows. Section II reviews work related to both assertions and testing. Section III describes the strategy used for combining assertions and bounded exhaustive testing, as well as providing a simple example illustrating the strategy. Section IV provides the details of the experiment. Section V gives the experimental results. In Section VI we will evaluate the results along with the weaknesses of our approach. Section VII describes the future work. Section VIII concludes the paper.

II. Background & Related Work

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

A. Assertions

Rosemblum states that although assertions have been determined to be a powerful tool for detection of software faults during the lifetime of a software system, it has seen little widespread use in practice. [3] He mentions that the techniques and tools which are developed for embedding assertions into the code have not been used because the programmers have little idea of what information should be specified in assertions and also the tools do not allow flexibility in customizing checks. There is also a description about the classification of assertions, meaning those which were the most effective at detecting errors. Rosenblum concludes that there is a need for more comprehensive, controlled experimental studies on larger systems with multiple developers for revealing the most effective techniques for using assertions to improve the quality and reliability of software systems.

Also there are a myriad of research projects which have tried to enhance the power of simple assertions [5], [6]. Meyer's design-by-contract [7] is one of the most famous ones. Meyer provides an approach for inserting assertions in object-oriented programs. He specifies that each method should have preconditions and postconditions which ensure the state of the input and output, respectively, of a given method. The assertions that deal with the class as a whole were referred to as class invariants. According to Meyer, these class invariants were properties that should always hold for a class. Ideally, it would be possible to prove mathematically that the assertions match the implementation of a method or class. Instead, Meyer says that we can use the assertions as runtime checks. We use assertions as a means of runtime checks as Meyer suggests. Since we have translated the formal specification into assertions, the assertions not only serve as validation of correctness but would also ideally serve as a verification that the specification matches the implementation. In our case, the formal assertions therefore act as test oracles, ensuring the validity of any output from the system.

B. Testing

Exhaustive testing has been proved infeasible and impractical [2]. Usually, the test data set is infinite or significantly large, which creates a situation where 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. [2]

TestEra by Khurshid and Marinov [8] is a novel framework for testing of Java programs. TestEra deals with the issue of making testing practical by generating a bounded test data set on being given a formal specification for a method. For each input in the bounded test data set, the method is run with this input, and the method postcondition is used as a test oracle to check the correctness of each input. In their future research, TestEra's authors plan to extend the capability of TestEra so that users can decide the point at which they think enough testing has been conducted. We will use bounded exhaustive testing instead of structural code coverage as discussed in TestEra.

TestEra is built upon Alloy and Alloy Analyzer(AA) with the aim of checking actual Java implementations. The basic framework for TestEra is demonstrated in Figure 1. TestEra generates three files on being given an Alloy specification. The first file is for generating inputs, and the second one is for checking the correctness. The third file consists of the java test cases.

The inputs are translated into the Java test cases and the program is then executed. During the first phase of the execution, Alloy Analyzer is used by TestEra to generate all non-isomorphic instances from the Alloy input specification. In the second phase, testing is done on each of the instances. The resultant output is again converted back to Alloy so that Alloy Analyzer can verify the input and output according to the correctness specification. A counterexample is reported if the test fails. If the test succeeds then another input instance is evaluated. Details of TestEra can be found in [8].

Other research has also shown that the high reliability of a system is proportional to the amount of testing. In their research, Butler and Finelli state that reliability growth models are portrayed to be incapable of overcoming the need for exorbitant amounts of testing. [9]

Pseudo-exhaustive testing has also been conducted in the hardware world, specifically in the study of VLSI chips. This type of testing is very similar to bounded exhaustive testing, where representative portions of the functional domain are tested without exhaustively covering all possibilities. The paper by Jone, Rao, and Chang describes the effective generation of pseudo-exhaustive test patterns for combinational VLSI circuits. [10]

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

III. Approach

This section will describe the manner in which we went about integrating formal assertions with bounded exhaustive testing. In the research, there were two distinct focuses. The first was to implement assertions derived from the formal specification. The second was to implement a modified version

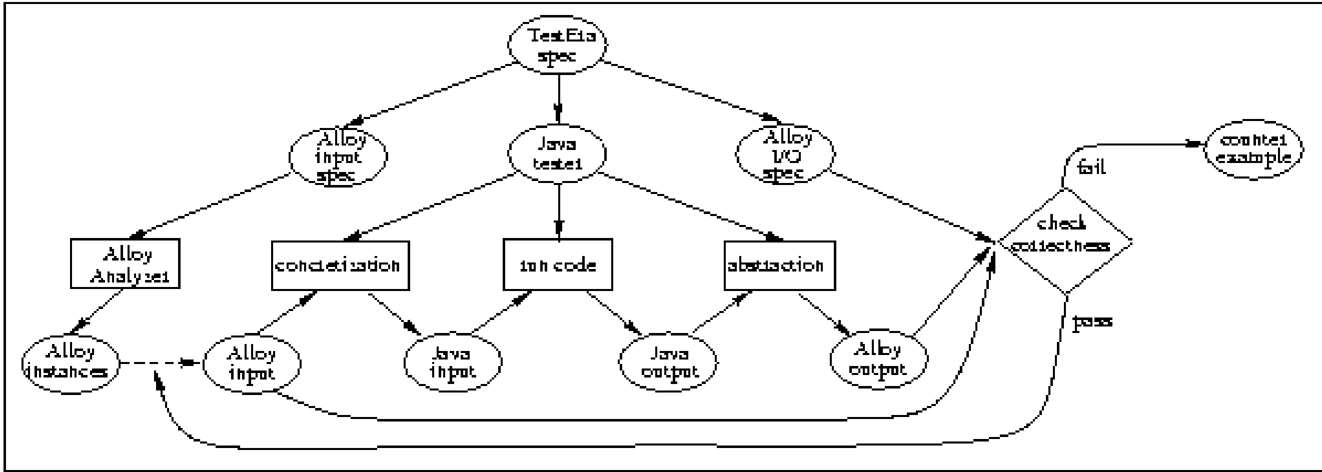

Fig. 1. Framework of TestEra [8]

of exhaustive testing, which will be referred to as bounded exhaustive testing. The final goal of the research was in allowing these two foci to work together to test the system.

A. Formal Specifications & Assertions

This section presents an example to illustrate how formal specification derived assertions can be combined with bounded exhaustive testing to further ensure the reliability of a software system. We will inspect a small system that simply sorts a list of items.

```
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, [11] a popular software specification language.

Sort

$Sort : \operatorname{seq} \mathbb{Z} \rightarrow \operatorname{seq} \mathbb{Z}$

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

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] > temp); 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.

B. Bounded Exhaustive Testing

The next step, in terms of our research, was bounded-exhaustive testing on the *Sort* algorithm above with the embedded assertions. Scope plays an important role in bounded-exhaustive testing. Scope can be thought of as looking at certain aspects or characteristics of the structure. In this case, scope could be the possible lengths of the *list*, the possible types of inputs for the *list*, or some other characteristic of the *list* one chooses to inspect. The idea behind bounded exhaustive testing is to find a way to enumerate all the

possible inputs of a method or system. The characteristics to be enumerated are the scopes of the model or system. Using the *Sort* algorithm example, enumeration could 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
   a) 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.
   b) 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 as is illustrated in Table I. For example, consider the *list* of length 3. The list could be $r = [x_1, x_2, x_3]$ with 3 distinct values, $r = [x_1, x_1, x_1]$ where there is only 1 distinct value, or some other combination of values.

Length of r	Distinct values in r
0	0
1	1
2	1,2
3	1,2,3

TABLE I
POSSIBLE VALUES IN THE *list*

IV. EXPERIMENT

We utilized Nova Solver [12], a reliability-testing software system, to assess the efficiency of our testing method. It is a software system with about 10000 lines of code. The Nova solver system has already undergone testing and has been in a functional state.

The Nova Solver takes a dynamic fault tree as its input, and computes a probability of failure for this fault tree through a number of conversions and calculations. Figure 2 shows an example dynamic fault tree, **DFT**. A dynamic fault tree contains a set of basic events, i.e. Event T, A, and S. These basic events are inputs to gates or functional dependencies. The gates can be AND, OR, PAND, Threshold, or Spare gates. For our research, it is only necessary to know the types of gates, not their functions. The functional dependency enforces a constraint on the system. The number in the circle adjacent to the Event S in Figure 2 defines the replication value of that particular event. The fault tree is passed to an instance of `Fault_Tree_Semantics`, which acts upon a given fault tree to create an object of type `Failure_Automaton`. Failure Automaton is comprised of a set of states and transitions. The set of states contains the information of every state of the system since its instantiation. The set of transitions demonstrates the transitions, i.e. *to* and *from*, of every state. This instance of `Failure_Automaton` is passed to `Failure_Automaton_Semantics`, which creates an instantiation of `Markov_Model`. Finally, the system reliability is computed from this markov chain. A detailed description about Nova Solver can be obtained at [12].

A. Formal Specifications & Assertions

In addition to knowledge of the general structure of the input to Nova Solver, it is important to note that the system has a well-defined formal specification which will be used for deriving formal assertions. The main goal of the research was to demonstrate that the new approach could even find bugs on a system that has been in use for a few years.

In order to implement assertions, we had to have knowledge of a few different entities. The first was the formal specification of the Nova Solver, which was written in Z. [11] A careful understanding of Nova Solver code was required to correctly implement the new assertions. This obstacle was resolved by conducting thorough code reviews and meeting the author of Nova Solver several times. However, before inserting these assertions into the code, it was necessary 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 was that there would be little to no change in the meaning and power of the assertion after it is converted. Also there were instances where we had to lock subsections of code before assertions could be used on that code. For example, consider an event which has 3 attributes where the values of these 3 attributes must add up to 1. The values are entered by the user after the instantiation of the event. The check cannot be performed immediately after the instantiation of the event since the values have not been entered by the user yet. Once the values are entered, a function can be called to do the assertions. Our first focus culminated with all the specification based assertions being added.

B. Bounded Exhaustive Testing

The second focus of the research was the idea of bounded exhaustive testing. The goal of that is to test substantial portions of the functional domain without testing all possible inputs. The input possibilities and formats are of extreme importance for our bounded exhaustive testing. Many of the possible inputs had previously been generated. The test cases consisted of all non-isomorphic test input up to a given bound. A unique component of the test cases was the bound for which the test cases were generated. The bound should be such that the probability of that event occurring becomes very unlikely after that bound. The bounded exhaustive testing and assertions were then be used in combination.

TestEra [8] software is used for automated testing of Java programs. For this research, we were able to automate the generation of the test cases for bounded exhaustive testing. TestEra generates inputs from an Alloy specification. We were able to specify the inputs to Nova Solver, dynamic fault trees, using Alloy and therefore generate all inputs up to a given bound using TestEra. We used TestEra for just generating the

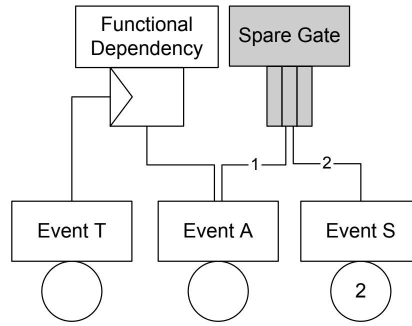

Fig. 2. Example dynamic fault tree

test cases. We did not use it to verify the output according to the correctness specification

Once all test case inputs are generated up to a certain bound, we ran the Nova Solver version containing formal assertions with each of the generated inputs. Because the formal assertions had been added to the system, it was no longer necessary to verify the correctness of the output from the system. For each input that successfully completed execution, we had verification that the system met the requirements of the formal specification based assertions. However, if an input failed a formal assertion, we had a concrete counterexample of an input which is not following the constraints of the specification. The latter case showed the presence of a bug in the system.

V. Experimental Results

This section provides the results from integrating formal assertions with bounded exhaustive testing on the Nova Solver system. Nova Solver was written primarily in C++, so the assertions derived from the formal specification were also written in C++. For the testing, we used the unit test suite for the Nova Solver initially, then we did used the test cases generated up to a given bound by TestEra [8].

During our initial review of the code we were able to point out certain inconsistencies in the code. The inconsistencies are as follows:

1) It was found that certain operator overloading defined in the specification was not implemented.
2) The other inconsistency was in the **Probability** constructor which takes a parameter of type `const long double`, but it uses the constructor of Real class which takes the parameter of type `const double`. So if the probability is very small it would be truncated to zero, which in case might generate incorrect reliability.
3) In an operator overload in **Coverage_Model**, there is an `if` statement that should logically never evaluate to *true*.
4) The last one was more of a bug than an inconsistency. The copy constructor in the **Fault_Tree** class did not include copying an attribute of the fault tree the (i.e. `system_event`).

A. Assertions

Next we present the data and results obtained during the initial phase of translating the specification into assertions. From the formal specification, we converted the constraints of the system into approximately 2300 lines of C++ code. The assertions added were most often non-trivial functions checking some constraint on the state of the system. This effort was completed by three programmers over a period of two months.

Once all the assertions had been derived from the formal specification and added to the system, we ran the test suite originally created for the system. The first implementation error involved a violation where *Time* objects were allowed to have values equal to 0. However, the specification only allows values of *Time* objects to be greater than 0. The unit test suite illustrated this bug once the formal assertions were integrated into the system. A similar violation was detected regarding *Thresholds*, which needed to be greater than 0. This assertion caused another test case from the original test suite to fail.

These bugs found were in the actual test cases. These were errors in the previously implemented test suite which had gone undetected. The fact that the assertions even found errors in the test cases gives major insight into the impact the formal assertions can have on the reliability of a system. The Nova Solver system was implemented in a way that did not always match the specification. Due to the fact that the system computes fault tree reliability correctly, it was not seen as a bug that the specification and code did not mirror the other. However, there were several instances where the assertions derived from the formal specification failed due to this lack of conformity. The majority of these failures came as a result of inadequately designed test cases. In the unit test suite, components of the system were being created and added to the system in such a way that the resulting system did not meet the constraints of the specification. The formal assertions caught these illegal states:

1) The **Failure_Automaton** was going from a `Failed_Uncovered` state to an `Operational` state, while the specification requires that a transition from a `Failed_Uncovered` state can only go to a `Failed_Uncovered` state.

2) A test case was making the `to_state` and `from_state` the same for a **Failure_Automaton_Transition**, which goes against the formal specification.
3) The specification requires that the `stateOfEvents` is equal to the last `history`, but the test case was creating a `stateOfEvents` that did not meet this criteria.
4) The test case inserted `states` and `transitions` into a **Markov_Model** object and created an illegal state for the **Markov_Model**.

The resolution of these bugs was the decision that a `finalized()` method needed to be called when a component of the system was completely built. From this `finalized()` method, all assertions corresponding to a component were called to verify the constructed component matches the specification. While this does not rule out temporary imbalance of the system, we concluded that it was sufficient to have certain checkpoints where the state of the system would be evaluated.

From this data, it is evident that the assertions helped us find bugs during the unit testing. We were able to find errors in the implementation, but additionally we found instances where the test cases were actually creating illegal objects and states. It also helped demonstrate how faulty the manual implementation of test cases could be.

B. Bounded Exhaustive Testing

This section presents the data gathered from using the formal assertions to check test cases generated using TestEra [8]. The idea of bounded exhaustive testing is to generate all possible inputs up to a certain small bound. Nova Solver takes a dynamic fault tree, **DFT**, as its input. Therefore, bounded exhaustive testing entailed creating all dynamic fault trees of a given size.

Because we are attempting to bound the size, we must look at certain characteristics of the dynamic fault tree to determine the bound. These characteristics of the model are called the scope. The policy is to generate all models which are less than or equal to the given bound. For our research, there were four characteristics used to define the scope, including the natural, injective sequence, event, and functional dependency. The functional dependency and event scopes limit the number of functional dependencies and events, respectively, in the model. The isequence scope limits the number of inputs a gate can have. The natural scope is the main scope which limits the thresholds and replications of the events in the model. We have currently generated and tested up through scope 3.

During bounded exhaustive testing, we found three main bugs in the system. Testing was done on both constrained and unconstrained DFTs. A constrained DFT is one that is connected, whereas an unconstrained DFT could possibly be disconnected. This is only important because Nova Solver supports both types of DFTs. An earlier implementation of a reliability solver, Galileo [13], supported only the constrained DFTs. At scope 2, none of the test cases caused a formal assertion to fail. In the constrained testing at scope 3, we found thousands of test cases failing. While testing unconstrained inputs of size 3, we again found thousands of failing test cases.

Main Scope	Test Cases	Failures	Type
natural2	56525	0	Unconstrained
natural3	2870095	18552	Unconstrained
natural3	48790	8260	Constrained

TABLE II
NUMBER OF TEST CASES VS. FAILURES

It should be noted here that all assertions that failed during the testing of the traditional test suite were removed before the bounded testing, as we were attempting only to find bugs which had not yet been reported.

Table II lists the number of test cases in relation to the formal assertions that failed during testing. For a couple of reasons, it was not surprising that the DFTs of size 2 did not fail any test case. First, the earlier bugs were already removed from the system. Secondly, a dynamic fault tree of scope 2 is almost trivial, thereby making it unlikely that this scope would produce any errors.

Although Table II shows over 25,000 failures, these only correspond to two distinct bugs in the software system. They are as follows:

1) **Probability** greater than one: There was an assertion failing because the the probability was greater than 1. Upon evaluation it was found that in those **Fault Trees**, every state was a failed state. The Nova Solver then added up all the probabilities, and they should have equaled 1. Due to numerical imprecision, they were actually slightly above 1.
2) Input Error: There was an error in *First_Occurrence_Time(const History&, const Event&, const Replication&)*. Basically, it was a copy-and-paste bug. The coder copied and pasted the lines of code, changed input1 to input2 in one place, and forgot to change it in the second place.

Table III presents the timings for the testing for each scope. The table gives the total number of distinct types tested at this scope, the total number of test cases for this whole scope, the total time for all test cases to run, and if the test cases were constrained or unconstrained. It is evident that the number of test cases and consequently the time necessary to run the tests is large for even the DFTs of size 3. It should be noted that these times are a total amount of time spent to run all test cases on a single Intel(R) Pentium(R) 4 3.00GHz processor. Each of the test cases could be run in parallel on a cluster or on a high speed processor to minimize the total time of testing by a significant magnitude. Much of this time for testing can be attributed to I/O costs.

VI. EVALUATION

In this section, we will evaluate the experiment and the method of combining formal assertions with bounded exhaustive testing.

From the experiment, we were able to find bugs that have thus far gone undetected after traditional testing and academic use of the software for a few years. We also saw a number

Main Scope	Sub-Scopes	Test Cases	Time	Type
natural2	13	56525	33.5 hrs	Unconstrained
natural3	19	2870095	346 hrs	Unconstrained
natural3	12	48790	11.7 hrs	Constrained

TABLE III
NUMBER OF TEST CASES VS. TIME

of important points regarding formal assertions and bounded testing. While we were able to add formal assertions to the system, the process was not trivial. Most of the translations from the specification did not turn into one line assertions in code. Instead, a single line from the specification was often a function in the code. This is the reason that the number of lines of code added was high, nearly one-fifth of the total system's code.

Secondly, it is not always possible to do this translation from the specification into assertions in the system. This conversion is dependent on how well the code mirrors the specification. If the system is different in any way, it can make the translation to code very difficult or even impossible. It is important to note that even a strict mirroring of the code to the formal specification does not imply the translation of the specifications will be a trivial matter. Although our focus was not on efficiency, it is possible that translating the specification into a runtime check may involve an algorithm that has a large complexity or is NP complete. In this situation, it is necessary to decide if reliability or efficiency is more important.

The case study helped us decipher more information about the bounds. For example, the functional domain is often infinite or sufficiently large. But the occurrence of errors becomes more and more rare as we start going away from the base case. At one point the probability of that error occurring will 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.

With bounded exhaustive testing, it was expensive to generate and test all inputs up to a given small bound. We were able to find more errors in the system through this testing, but the time needed for this testing is significant enough that it may only be suitable for systems whose reliability is of grave importance. For example, it may not be worthwhile for making a game or a word processor. Since the initial testing, a number of optimizations have been added that can currently lower the total time needed by a factor of 10. However, these optimizations were made to Nova Solver, so the time improvement was system specific.

After adding the assertions into the system, the test suite revealed a few bugs. The new version of the system with assertions added was useful in showing flaws in the test suite. Many of the test cases were creating components of the system illegally or doing flawed computation. The improvement of the test suite was a positive side-effect that we had not originally expected. This may well be an area of further research.

In terms of our experiment, we only tested up to scope 3. In order to know how many bugs can be found through bounded exhaustive testing, it would be necessary to test up to a higher bound. We also ran most of the test cases without optimization flags. In order to know a better estimate for time, the test cases would all be run with the optimization flags. In this way, we would be able to know more definitely if this method of integrating formal assertions with bounded testing is practical.

VII. FUTURE WORK

In our experiment, the combination of formal assertions and bounded exhaustive testing was done on a previously developed and tested system. A complete evaluation of the approach would involve using it on a real system as it is being developed. Such an approach would identify by what magnitude our method surpasses the existing testing methodologies, especially the manual generation of test cases.

In terms of errors the case study can give us insights about the 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. This leads to the issue of identifying the right bound for exhaustive testing. Is it sufficient to stop testing once we have stopped finding bugs on a specific bound or run a few million test cases? More research has to be conducted to derive better bounds for exhaustively testing the system which would be dependent of the distribution of remaining errors.

Assertions may hurt the efficiency and run time of the programs. As discussed earlier in the evaluation some of the runtime checks may involve an algorithm that has a large time complexity or even is NP complete. More research has to be conducted so that assertions do not cripple the efficiency of a system.

Assertions have to be tweaked so that they avoid numerical imprecision. The program may be correct but due to the accumulation of minute numerical errors the computation may appear to be incorrect. Special resolutions would need to be adapted for those types of cases.

VIII. CONCLUSION

The integration of assertions and bounded exhaustive testing is a novel approach for testing the software. We were able to achieve favorable initial results, where bugs were found through our integrated approach that were previously undetected. However, the time spent on writing formal specification based assertions and conducting bounded exhaustive testing may not be pragmatic for non-critical systems. Thus, our research seeks to bolster the reliability of critical systems, whose failure can result in loss of time, revenue, or life.

Acknowledgments

We would like to acknowledge Dr. Coppit for his contribution to our research. We also want to extend our gratitude to him for letting us use his Nova Solver as our case study. An addition acknowledgment goes out to Ms. Jennifer Haddox-Schatz for her collaboration in the adding of formal assertions to the Nova Solver software system.

References

[1] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, Oct. 1972.

[2] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky, "Validation, verification, and testing of computer software," *ACM*, vol. 14, no. 2, June 1982.

[3] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, Jan. 1995.

[4] P. E. Ammann, S. S. Brilliant, and J. C. Knight, "The effect of imperfect error detection on reliability assessment via life testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 2, pp. 142–8, Feb. 1994.

[5] D. C. Luckham and F. W. von Henke, "Overview of Anna, a specification language for Ada," *IEEE Software*, vol. 2, no. 2, pp. 9–224, Mar. 1985.

[6] J. M. Voas and K. W. Miller, "Putting assertions in their place," in *Proceedings of the International Symposium on Software Reliability Engineering*. Monterey, CA: IEEE, 6–9 Nov. 1994.

[7] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice Hall, 1997.

[8] D. Marinov and S. Khurshid, "TestEra: A novel framework for automated testing of Java programs," in *Proceedings of the 16th IEEE Conference on Automated Software Engineering (ASE 2001)*. San Diego, CA: IEEE, 26–29 Nov. 2001.

[9] R. W. Butler and G. B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 3–12, Jan. 1993.

[10] W. B. Jone, J. C. Rau, and S. C. Chang, "A tree-structured lfsr synthesis scheme for pseudo-exhaustive testing of vlsi circuits," *IEEE*, Oct. 1998.

[11] J. M. Spivey, *The Z Notation: A Reference Manual*, 2nd ed. Prentice Hall International Series in Computer Science, 1992.

[12] D. Coppit, "Engineering modeling and analysis: Sound methods and effective tools," Ph.D. dissertation, The University of Virginia, Charlottesville, Virginia, Jan. 2003, uRL: http://www.cs.wm.edu/~coppit/papers/dissertation.pdf.

[13] K. J. Sullivan, J. B. Dugan, J. Knight, *et al.*, "Galileo: An advanced fault tree analysis tool," 1997, uRL: http://www.cs.virginia.edu/~ftree/index.html. [Online]. Available: http://www.cs.virginia.edu/~ftree/index.html