Design-Time Detection of Implementation Errors Using Formal Code Specification

Iman Saleh, Gregory Kulczycki

Abstract— In this study, we are investigating the effect of formal methods on writing bug-free programs by measuring the ability of detecting design-time errors based on how completely the code was formally specified. We analyzed a collection of 13 C# classes that had been formally specified in Spec# by previous researchers. By eliminating various portions of these specifications, we came up with five distinct levels of specification: (1) no specification, (2) non-null specification, (3) non-null + assertions + invariants, (4) non-null +preconditions, and (5) all specifications. We adapted a mutation methodology to inject errors into the implementations of these classes and analyzed how many of these errors were caught using the Boogie verifier for Spec#. We found that Boogie was able to detect significantly more errors for the fully specified classes.

Index Terms—Error-Checking, Specification, Verification

1 Introduction

In this work, we investigate how formal specification can be employed to detect implementation errors. We use automated verification and reasoning techniques to achieve that goal. Our approach provides an insight on the usefulness of different specification constructs on improving the code implementation practices and potentially increase software correctness. Our approach also aims at discovering the benefits and shortcomings of using currently-available realizations of automatic code verification tools. An experimental methodology is described for investigating and validating this research opportunity. We adapt a methodology from the software testing field where code mutation is used to assess the quality of a testing technique. Mutation testing is carried on by injecting errors in the code and measuring the ability of a testing tool in detecting these errors. The main assumption here is that the number of mutation errors detected by a tool is an indication of number of errors that this tool can detect in the future when unknown bugs are present in the code. We use a similar methodology to evaluate the ability of code specification in detecting errors.

2 Research Hypothesis

We are interested in studying the effect of using formal code specification and verification in enabling writing bug-free code. Our research hypothesis can hence be stated as follows: "The formal code specification and verification facilitates the discovery of implementation errors at design time. These errors cannot be detected using a non-verifying compiler".

• *Iman Saleh and Gregory Kulczycki are with the Computer Science and Applications Department, Virginia Tech University, Falls Church, VA 22043. E-mail: {isaleh,gregwk}@vt.edu*

Manuscript received (insert date of submission if desired). Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

3 Evaluation Methodology

3.1 Data Set

We consider a set of 13 formally-specified classes. These classes implement a collection of textbook algorithms from [1], including search and sort algorithms, basic data structures, mathematical calculations and array manipulation.The classes are implemented and specified by the authors of [2] using C# as the programming language, and Spec# [3] as the formal specification language. The classes represent a set of simple yet practical examples of using code specifications.

3.2 Steps

We test our hypothesis by applying the following steps:
1. Each class in our data set is verified using the Spec# static code verifier, Boogie [3], to ensure that the implementation initially satisfies the formal specifications.
2. A fault injection tool implemented by the authors of [4] is used to automatically introduce errors in each class. This step simulates programmers' errors by randomly applying mutation operators. The mutation operators used in our experiment are described in Table 2.
3. The Boogie verifier is executed on each mutant of each class. The verifier output is investigated to check if it detects the injected error.
4. Step (3) is repeated for different types of errors. The total number of errors detected is calculated at each of the following specification levels:
- L0: No specification, this level acts as a baseline
- L1: Non-null types
- L2: Non-null types + loop and class invariants
- L3: Non-null types + preconditions
- L4: Full specification

These levels are selected from a practicality standpoint as they capture the levels of efforts that can be invested by programmers in writing formal specifications.

xxxx-xxxx/0x/$xx.00 © 200x IEEE

TABLE 2
MUTATION OPERATORS USED IN THE EXPERIMENT

No.	Operator	Description	Example
1	AOR	Arithmetic Operator Replacement	a = b + c to a = b – c
2	ROR	Relational Operator Replacement	while(a < b) to while(a > b)
3	PRV	Reference assignment with other compatible type	a = b to a = c
4	EOC	Replace == with Equals()	x == 0 to x.Equals(0)
5	JID	Member variable initialization deletion	int[] a = new int[2] to int[] a
6	JTD	*This* keyword deletion	This.x to x

3.3 Evaluation Metrics

For each of the five specification levels described earlier, we measure the *mutation score*, which is the percentage of errors detected to the total number of errors injected into the code. The main idea is that the number of errors detected using either of the specification levels studied in our experiment is an indication of the correctness of software produced using that level. To evaluate performace, we also measure the *verification time* taken by the Boogie verifier at each level of specification.

3.4 Results and Analysis

A total of 248 mutants were generated and formally verified throughout the experiment. The mutation scores are depicted in Figure 1 using a boxplot (see appendix).

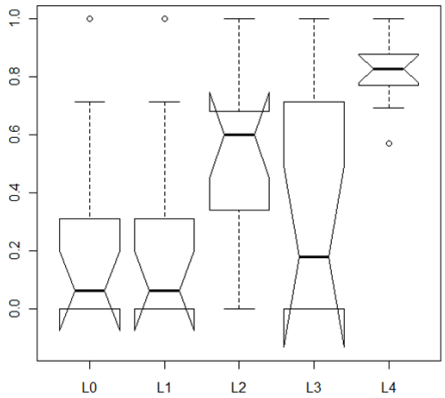

Fig. 1. The mutation scores achieved at different specification levels

We can draw the following conclusions from our analysis of the results:

- The mutation score is significantly high for the full specification (83%, on average), followed by the use of invariants (60% on average).
- The non-null types specification does not enhance the error detection score.
- The preconditions ability in detecting errors is very depending on the program and needs further testing.
- We have investigated a case where the same code is specified using two different loop invariants. The first invariant uses relational operators, while the second relies on some Spec# keywords like *max* and *min*. The second invariant enables the detection of more errors. This issue needs more investigation.
- In some cases, the loop invariants have actually concealed an error that is detected when no invariant is added. This is due to the way the Boogie verifier works.

It should be noted here that the mutation tool generates errors that are syntactically correct and hence none of these injected errors are detected by the C# non-verifying compiler. In other words, the mutation score of the non-verifying compiler is consistently equal to zero.

Our tests show no significant difference in verification times between difference specification levels. The time ranges between 0.08 to 0.11 seconds per program. Depending on the development environment, these numbers can be useful in estimating the overhead of using a verifying-compiler versus a non-verifying one.

4 CONCLUSION

In this study, we have provided evidence that formal methods can enable the detection of programmer's errors at design-time. Even though the full specifications in our experiments are not guaranteed to be exhaustive, these specifications have enabled the discovery of high number of errors injected in the code. Some errors are detected by simply using a verifying-compiler without adding code specifications. This is due to the fact that a verifying-compiler applies some additional checks, e.g. array bound checking and possible divisions by zero. A practicality issue that needs further investigation is the case where partial specification may generate errors due to the fact that the verifier cannot prove some of the assertions with partial specifications regardless of actual errors in the code.

APPENDIX: BOXPLOT

A boxplot [5] is a graphical way for depicting a set of data values. The bottom of the box is the 25th percentile and the top of the box the 75th percentile. The line across the middle of the box is the median, or 50th percentile.The plot also display outliers, which is a value that is significantly distant from the rest of the data. The boxplot is used to visualize the differences/similarities between data sets.

REFERENCES

[1] E.W. Dijkstra, W.H.J. Feijen, and J. Sterringa, *A Method of Programming*, Addison-Wesley, 1988.

[2] K.R. M. Leino and R. Monahan, "Automatic verification of textbook programs that use comprehensions," *Formal Techniques for Java-like Programs*, Berlin, Germany: 2007.

[3] M. Barnett, K. Rustan, M. Leino, and W. Schulte, "The Spec# programming system: An overview," *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, vol. 3362, 2004, pp. 49--69.

[4] A. Derezińska, "Advanced mutation operators applicable in C# programs," *Software Engineering Techniques: Design for Quality*, 2007, pp. 283-288.

[5] J.W. Tukey, *Exploratory Data Analysis*, Addison Wesley, 1977.