2015 IEEE 16th International Symposium on High Assurance Systems Engineering

Using Pairwise Testing to Verify Automatically-Generated Formal Specifications

Salamah Salamah, Omar Ochoa, and Yadira Jacquez
Computer Science Department.
University of Texas at El Paso
El Paso, Texas, USA

Abstract

In this paper, we report on the effectiveness of the testing approach known as pairwise or orthogonal testing in verifying the correctness of the LTL specifications generated by the PROperty SPECification (Prospec) tool. This tool assists the user in generating a large number (over 34,000) of formal specifications in formal languages, including Linear Temporal Logic (LTL). Pairwise testing is a technique that aims at, significantly, reducing the amount of test cases required for testing a particular software system while providing assurance of adequate coverage of the problem space.

Keywords—Formal specifications, LTL, Pattern, Scope, Composite Propositions, Pairwise Testing.

I. INTRODUCTION

Formal verification techniques such as model checking [1], theorem proving [2] and runtime monitoring [3] have been effective in improving dependability of wide range of software systems. Critical to the effectiveness of the formal verification approaches are the quality of the formal specifications of system properties. The PROperty SPECification (Prospec) tool [4] assists users in defining formal specifications in multiple languages. While Prospec generates a large number of formal specification (over 34,000), it is imperative that these specification are validated and tested to make sure they match the original intent of the developer. The large number of specifications supported by Prospec, increases the need for an efficient method of validating the correctness of the generated specifications. Pairwise testing [5] is a technique that significantly reduces the number of test cases used in the verification of a particular problem space.

The work in [6] defined templates to generate a wide range of formal specification in Linear Temporal Logic (LTL). These templates have been validated using formal proofs and software inspections. However, the Prospec tool's implementation of these templates has not been adequetly tested to ensure the adherence of the generated formulas to the formal requirements as represented in the defined templates. A major issue in testing the Prospec-generated LTL formulas is the share volume of formulas. This is the main motivation for using pairwise testing to reduce the number of formulas to be tested. The technieque is used to reduce the number of formulas to be tested and offer assurance that the set of formulas put under test provides adequate coverage of all formulas produced by the tool.

Pairwise testing is a combinatorial testing technique that seeks to assure that test cases focuses on defining test sets that provide tests of all pairs of variables instead of tests that combine all variables. The formal definition of pairwise testing strategy is as follows: Given a set of N independent test variables v1, v2,..., vN with each variables vi having Li possibilities = {li,1,..., li,Li}. The set of tests R produced contains N test levels, one for each test-factor vi; and, collectively, all tests in R cover all possible pairs of test-factor levels (belonging to different parameters); in other words, for each pair of factor levels li,p and lj,q where $1 \le p \le L$i, $1 \le q \le L$j, and $i \ne j$, there exists at least one test in R that contains both li,p and lj,q.

In a hypothetical system with three input variables *a, b,* and *c* each of which can assume the values of 1 or 2 pairwise testing requires four test cases compared to eight required for exhaustive testing.

In this work, we used pairwise testing to reduce the number of formulas to be tested for the *Response (Q, P)* within the scope *Between L and R* [6]. In [6] we showed that this combination of pattern, scope, and Composite Propositions (CPs) yields a total of 6,561 total formulas. Using pairwise testing we are able to test this combination using only 78 formulas. In generating the list of combinations tested using pairwise technique for the *Response (Q, P)* within the scope *Between L and R* we used the variables *Q, P, L,* and *R*, as the variables of interest. Each of these variables had 11 possible CP values as described in [6].

II. TESTING EXPIREMENT SETUP

In order to test the formulas generated by the Prospec tool we 1) used the web tool Hexawise [7] to produce a combinations of formulas to be tested - the tool suggested 78 formulas to represent the set of 6,561 total formulas for the pattern/scope/CPs combinations, 2) tested each of the 78 formulas using the notion of traces of computations [7], equivalence classes, and boundary testing techniques to test the behaviors accepted and rejected by the suggested 78 formulas, 3) used the results of testing to fix the rest of the formulas, and 4) used Hexawise to define another, disjoint, set of 78 formulas for the same pattern/scope/CPs combination and performed regression testing to examine if the fixes do hold and whether or not they have introduced new defects.

978-1-4799-8111-3/15 $31.00 © 2015 IEEE
DOI 10.1109/HASE.2015.46

III. EXPIREMENT RESULTS AND ANALYSIS

In testing the formulas we found two types of defects. The first of these defects was in all formulas where the parameter *R* was represented with a CP class of type *Event* [6]. Such formulas always returned *valid* no matter what the trace was. This was the case for 39 out of the 78 (50%) of the formulas tested. In examining the tested formulas, we discovered that the code used by the Prospec tool to generate these formulas had a missing pair of open-close parentheses. We noticed that such formulas have the following structure:

G (L & !R) → rest of formula, where *L* is the LTL formula representing the CP for *L* and *R* is the LTL formula representing the CP for *R*. Because the *Always* (G) operator in LTL has a higher precedence than the *imply (→)* operator, the specification as it is generated by Prospec states that "*if it is the case that L and !R always hold then the rest of the formula must hold"*. This is obviously not the original intent which is "*it is always the case that when L and !R hold, the rest of the formula must hold"*. To correct this defect, it is necessary to add an open parenthesis immediately after the first *Always* operator and a close parenthesis immediately before the first *imply* operator.

The second defect appeared to effect formulas where parameter *L* was of type *EventualE* and *R* is of any Condition Type. This defect affected six of the 78 formulas. The defect was similar to the earlier type in that it was caused by a missing pair of open-close parenthesis as in the following example where *L* is of type *EventualE*, *R* is of type *ParallelC, P* is of type *EventualC* and Q is of type *ConsecutiveC*. Examples of the corrected formulas for defects 1 and 2 are shown in the next two boxes.

(G((((!a & !b & !c) & ((!a & !b & !c) U (a | b | c))) & !(!d & !e & !f & X (d | e | f)))) → ((!a & !b & !c) & ((!a & !b & !c) U ((a | b | c) & (!((!(!d & !e & !f & X (d | e | f)) U ((((!h & !i & !k) & (!(!d & !e & !f & X (d | e | f)))) & (((!h & !i & !k) & (!(!d & !e & !f & X (d | e | f)))) U (((h | i | k) & ((!((l | m | n) & (!(!d & !e & !f & X (d | e | f))))) U (!d & !e & !f & X (d | e | f)))) & (!(!d & !e & !f & X (d | e | f))))))))))))

(G(((!a & !b & !c) & ((!a & !b & !c) U (a & !b & !c & X(((!b & !c)U (b & !c & X(!c U ((c & (!(d & e & f)))))))))) -> ((!a & !b & !c) & ((!a & !b & !c) U (a & !b & !c & X(!b & !c U (b & !c & X(!c U ((c & (!((!(d & e & f)) U ((h & (!(d & e & f))) & (X ((!i & (!(d & e & f))) U ((i & (!(d & e & f))) & (X ((!k & (!(d & e & f))) U ((k & ((!((l & (!(d & e & f))) & (X ((m & (!(d & e & f))) & (X (n & (!(d & e & f))))))) U (d & e & f))) & (!(d & e & f))))))))))))))))))))

To gain further assurance of the coverage of the set of test formulas generated by the pairwise testing technique, we used Hexawise to generate another disjoint set of 78 formulas and we used similar test cases to test these formulas. The two defects described above were also apparent in the new set of formulas. The first defect affected 49 of the new 78 formulas while the second defect affected four of the 78. Similarly the corrections made earlier were enough to fix all 49 formulas affected by the first defect and all four affected by the second defect.

IV. CONCLUSION AND FUTURE WORK

Although the use of formal methods in software development has shown potential in increasing the dependability of the developed systems, there is still reluctance to using formal techniques in software assurance. One of the issues in employing formal verification techniques is the difficulty in writing, reading, and understanding formal specifications. Because of the increased use of formal verification techniques in the software development there is a need for software engineers to be able to validate formal specifications and gain confidence in their generated software properties.

The effectiveness of the formal verification approaches depends greatly the on the correctness of the formal specifications of properties that are a major component of these techniques. While Systems such as the Specification Patterns System [8] and Prospec [4] provide support for the generation of formal software properties that can be used in formal verification tools such as the model checking and runtime monitoring, it is imperative that the generated formal specifications do indeed match the user's original intent. In this work, we showed how testing techniques such as pairwise testing, equivalence partitioning and boundary value analysis can assist in testing the correctness of Prospec's generated formulas. Using these techniques allowed for the testing of a much more manageable set of test formulas as well as test cases for these formulas.

While this work focused on testing the formulas for the *Response* pattern within the *Between L and R* scope, as a future work, we aim at replicating the testing approach to test formulas for all patterns, scopes, and CP classes. In addition, using 2-way combinations of pairwise testing may miss 10% to 40% or more of system bugs [8], and is thus not sufficient for safety-critical systems. As such we will attempt to use higher levels of combinatorial testing such as 4-way testing.

V. REFERENCES

[1] Holzmann G. J. The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Professional, 2004.

[2] Rushby, J., "Theorem Proving for Verification," Modelling and Verification of Parallel Processes, June 2000

[3] Stolz, V. and E. Bodden, "Temporal Assertions using AspectJ", Fifth Workshop on Runtime Verification, July 2005.

[4] Mondragon, O., A. Q. Gates, and S. Roach, "Prospec: Support for Elicitation and Formal Specification of Software Properties," Proceedings of 4th Runtime Verification Workshop, Barcelona, Spain, April 2004.

[5] Cohen, D, M., Dalal, S, R., Parelius, J., and Patton, G, C., "The Combinatorial Design Approach to Automatic Test Generation", In IEEESoftware, September 1996, 83-87.

[6] Salamah, S., Gates, A., and Kreinovich, V., "Validated Templates for Specification of Complex LTL Specifications" in the Journal of Systems and Software, Volume 85 Issue 8, August, 2012.

[7] Hexawise, https://www.hexawise.com/ accessed November, 2014.

[8] Specification Patterns System, http://patterns.projects.cis.ksu.edu accessed September 2014.

[9] Kuhs, R., Lei, Y., and Kacker, R., "Practical Combinatorial Testing: Beyond Pariwise". In Journal of IT Professional, volume 10, Issue, 3, 2008.