Using Combinatorial Testing to Reduce Software Rework

Redge Bartholomew, Rockwell Collins

Abstract. In developing many safety-critical, embedded systems, rework to fix software defects detected late in the test phase is the largest single cause of cost overrun and schedule delay. Typically, these defects involve the interactions among no more than 6 variables, suggesting that 6-way combinatorial tests could detect them much earlier. NIST developed an approach to automatically generating, executing, and analyzing such tests. This paper describes an industry proof-of-concept demonstration to see if this approach could significantly reduce the number of defects that escape into the test and evaluation phase of safety-critical embedded systems.

Introduction

Studies of safety-critical, embedded systems have shown that the rework required to fix late-detected software defects is one of the largest single components of their development cost and schedule—e.g., [1][2][3][4][5]. They also show that detection of these latent defects accelerates during late-stage testing and that those detected during operational test and evaluation have become more than just problematic. Much of this is attributable to verification tools and techniques that are becoming increasingly inadequate as the scale and complexity of software continues to increase [6][7][8][9]. An emerging need to develop parallel software for embedded multicore processors will make this problem worse [10]. Improvement requires tools and methods that prevent defect injection or that accelerate detection. They must do so, however, without a prohibitively large impact on normal development.

A study conducted by NIST and NASA looked at software defects detected over a 15-year period [11]. Systems studied included avionics, medical devices, web browsers, servers, space systems, and network security systems, and ranged in size from tens of thousands to hundreds of thousands of lines of code. It found that defects were triggered by the interactions among no more than six variables. This being the case, 6-way combinatorial test vectors might be able to detect them. Subsequently, NIST and the University of Texas-Arlington found an efficient algorithm for minimizing the number of test vectors that would cover up to 6-way combinations of input values [12][13][14][15]. They implemented this algorithm in a tool called Automated Combinatorial Test System (ACTS)[1].

The tool was effective at triggering defects, but verification testing required expected outputs, not just inputs, and creating these manually for thousands of inputs would be prohibitively expensive. NIST found an approach to automating this process using a model checker's counter examples. It also created a utility that merged the input vectors with their expected outputs as well as a test harness that read complete test cases, executed tests, analyzed results (compared actual versus expected outputs), and identified anomalies [16].

This paper describes an industry proof-of-concept study that used NIST's approach to automate unit testing of a software defined radio's control software. The goal was to determine if the NIST approach could cost-effectively reduce the number of latent software defects escaping into system testing and at the same time achieve the structural coverage required by regulatory authorities.

The Test Environment

Tests were generated, executed, and analyzed on a Windows 7, quad-core, 2.5 GHz, i5 laptop with 4GB memory. ACTS was used to create a model of the input variables, generate 6-way combinatorial test vectors, and export them to a networked server. The NuSMV[2] model checker generated the state space and exported it to the same networked server. An in-house utility function read the two files, searched the state space for states containing the ACTS vectors, reformatted them, and exported them as test cases back to the server. A commercial test harness, VectorCAST, instrumented the source code to track structural coverage, measured code complexity, imported the test case file, loaded test values into input variables, and executed tests. It also accumulated the achieved modified condition/decision coverage (MC/DC) [18], collected output variable values, compared actual with expected values, and identified discrepancies.

The code being tested was a software defined radio's control interface, containing 196,000 executable source lines of C++ code. The initial focus of the study was a code unit responsible for controlling the radio's waveform mode (e.g., HAVEQUICK, SINCGARS, Link 4) and operational state (e.g., idle, ready, running). This had 579 lines of code, 34 input variables, and 4 output variables of interest, used by 47 decisions nested up to 8-levels deep, spread over a 6-case switch. Its measured complexity (number of unique execution paths) was 46. In addition to the mode and state controller, the study tested another 70 of 717 code files.

Defining the Input Space

Developers provide ACTS with a name, a data type, and a set of values for each input variable. They also select the combinatorial strength of the vector generation (2-way through 6-way). ACTS then generates a set of input vectors containing all combinations of input variable values for the selected strength. Table 1 shows the 2-way vectors ACTS generated for the function:

```
if (c == true)
    e = a + b;
else
    e = a * d;
return e;
```

Defining the input space to maximize defect detection and structural coverage without significant test iteration (test, measure coverage, determine coverage gaps, add input vectors, repeat) is nontrivial [12]. The greater the number of input test values, the greater the code coverage but also the greater the likelihood of com-

	a	b	c	d
1	0	255	true	-1
2	0	256	false	0
3	0	255	false	1
4	15	256	true	-1
5	15	255	true	0
6	15	256	false	1
7	16	255	false	-1
8	16	256	true	0
9	16	255	true	1

Table 1: Two-Way Combinatorial Vectors

binatorial explosion. The smaller the number, the greater is the likelihood of missed defects and inadequate structural coverage.

A compromise is to limit input values to those representing equivalence classes [16]. For each input variable, possible values are segregated into groups that would ostensibly produce no difference of interest in code behavior or output value. One or more representative values are then picked from each group. This typically includes values that test behavior across instruction and memory architecture boundaries (e.g., positive and negative minimum and maximum values, and 0), data definition ranges, coordinate systems, units of measure, and so on, and also those that drive decision conditions.

Identifying representative values for boundary values was straightforward. Finding values for condition variables in complex, nested logic—values that would force the execution paths required for code coverage—took more time. MC/DC requires that every condition in a decision has taken all possible outcomes at least once, and that each condition in each decision has been shown to independently affect that decision's outcome. Demonstrating independence-of-outcome typically requires modifying each condition in a decision while all others remain fixed, and showing that this modification has changed the outcome of the decision. For the while-loop in

```
if ((a != b) && (a != c))
{
  …
  while ((a != b) && (a != c))
  {
    a = chan ();
  }
}
```

tests must be run to show that when both conditions are true, the loop is executed, and that when each is false but the other true, the loop is not executed. To determine the input space, values that force execution of each such path under the required conditions must be selected for each variable of each condition of each decision.

Enabling those values was difficult when the condition variable was an input and the values had to be loaded by an external procedure invoked from within a decision. In the example, the loop decision must be tested when $a = b$ and when $a = c$, neither of which conditions can be created by direct input from a test case. The value of *a* must be changed at runtime by the call to the external procedure *chan ()*, which is stubbed-out for unit test. The work-around was to add test-unique variables to the test cases generated by ACTS and the model checker. Test stubs were replaced with small procedures that loaded the value of the test-unique variable directly or indirectly into the condition variable. In the example, the test variable's value would be loaded into the return value of *chan ()*.

Generating a state space for all 34 input variables of the mode-state controller produced combinatorial explosion. Several separate sets of test vectors had to be generated instead, each set covering only those variables that interact to produce an output. The test harness assigned default values to those variables not included in a test case. Maximizing structural coverage required running all such sets of tests. In no case, however, was there an output value affected by interactions among more than six input variables, and in aggregate all 6-way combinations of interacting variables were tested.

Generating Expected Outputs and Executing Tests

The model checker is given a model containing variable definitions, their relationships, their values in an initial state, and how their values are determined in subsequent states. It then generates the state space (or a binary decision diagram of it), each state mapping a combination of input variable values to output variable values. See Fig. 1 showing the mapping of the input values from Table 1 to the output variable, *e*. For all states in which the value of *c* is *true*, the value of *e* will be equal to the value of *a* plus the value of *b*, which is expressed as $c = true : a + b$. In all other states, the value of *e* will be equal to the value of *a* times the value of *d*, expressed as $TRUE : a * d$. Fig. 1b shows a segment of the generated state space—the value of *e* followed by the input values that produced it.

In the NIST approach, the process of creating expected outputs for an input test vector relies on a model checker's counter-examples [17]. Ordinarily, to verify requirements or a design, developers using a model checker would create a model like the one in Fig. 1a , but they would also write properties the model must preserve—e.g., there must always be a way for the variable e to be 0, there must always be a way for it to be 272. The model checker attempts to prove that the model preserves these properties. Where it finds a violation of a property (a counter-example—e.g., an execution path in which e can never be 0), it produces a trace of the states that led to the violation.

To have a model checker determine an expected output for a given input vector, developers could negate a property and use the counter example to trace back to the input values that produced it. For example, they could specify that the variable e must never be 0. The model checker would detect a state that violated this property and generate a counter example showing the state transitions from the initial input values (the input vector) to the point at which e became 0. A simple utility could create a complete test case from a counter-example by merging the value of the output variable with the values of the input variables that produced it [16].

This study used a slightly different approach, requiring a smaller learning curve. Instead of searching through counter examples generated by the model checker, the utility function searches for each input vector across the entire state space generated by the model checker. The model in Fig. 1a generated 36 states: those containing all possible combinations of variable values. As shown in Table 1, all 2-way combinations of inputs can be covered by the nine input vectors generated by ACTS. The utility function finds state 4 containing the input vector, {0,255,false,1}, eliminates any irrelevant inputs and outputs from the state, reformats the remainder (the input vector and its expected outputs), and exports the result, {0,0,255,false,1}, to the test harness. When it has found and exported all 9 test cases, it is finished.

Developers then load the test harness with both the source code and the test cases, and map the test case entries to input and output variable names—e.g., map the first entry of the input

```
MODULE main
VAR
 a : {0,15,16};
 b : {255,256};
 c : {true,false};
 d : {-1,0,1};

DEFINE
 e :=
  case
    (c = true) : a + b;
    TRUE : a * d;
    esac;
```

Fig. 1a. NuSMV Model

```
------- State   4 ------
 e = 0
 a = 0
 b = 255
 c = false
 d = 1
------- State   5 ------
 e = 272
 a = 16
 b = 256
 c = true
 d = 1
------- State   6 ------
```

Fig 1b. State Space Segment

test case in Fig. 1b (0) to the source code variable e, the second entry (0) to the variable a. They can then execute the tests. Failures and the achieved code coverage can be monitored in test harness windows. Correctness of the expected outputs (verifying the oracle) is established when the resulting test cases are able to detect all seeded defects with no false positives.

Results

Putting aside defective or incomplete requirements, misinterpretations of requirements and design decisions, and other errors not revealed by exercising the code, at issue was whether such an automated test approach could cost effectively detect all (or nearly all) implementation defects. Evaluation criteria included accuracy, structural coverage, scalability, execution time, maturity, ease of learning, and ease of use.

Accuracy was measured in two ways: as the percent of seeded defects the tests detected; and as the percent of false detections (number of false positive detections as a percent of total detections). Defects were manually and arbitrarily seeded into versions of the code by changing values in arithmetic and logic statements, changing arithmetic signs, reversing and negating comparisons, deleting statements, and so on. In all, there were over 200. After debugging the NuSMV model, the search-export utility, and the test harness definition, the generated tests triggered all defects with no false detections.

The initial set of tests achieved 75% statement coverage, 71% branch coverage, and 68% MC/DC. The relatively low initial coverage was the result of the inadequately defined input space, described earlier. With a better understanding of how the input space was to be defined, the subsequently generated test cases achieved 100% MC/DC.

Scalability was an evaluation of both size (in this case, the number of input and output variables) and logical complexity. As mentioned earlier, after limiting inputs to only interacting variables, test generation never again produced state space explosion. After using test variables to deal with loops that changed the value of their condition variables, there were no further complexity issues.

Execution time was acceptable: for the largest vector generation model (19 input variables, 1 output variable), ACTS produced 2775 input vectors in six seconds, NuSMV generated the state space in about 60 minutes, and searching it and building the test cases took just over eight minutes. The test harness imported them in 15 seconds, created their executable tests in 12 seconds, and executed and analyzed them in under eight minutes.

Cost effectiveness was a measure of the value-in-use (accuracy, coverage, scalability, and performance), the effort required to learn the approach, and the effort required to use it on an ongoing basis. Learning to use ACTS was simple. NIST provides a tutorial that takes about two hours to process and contains everything needed to begin using the tool. Initial definition of the 34 input variables used by the mode controller took four hours, including initial equivalence class determination and value selection. Using the .pdf tutorial from the NuSMV web site, learning to develop NuSMV models and to use the NuSMV simulator to generate the state space took 20 hours. After encountering state space explosion, generating sets of input vectors for only interacting variables and selecting equivalence class values to achieve 100% branch coverage took an additional 16 hours. Finding a way of achieving 100% MC/DC coverage without manual intervention took another 16 hours. In total, the learning curve was 84 hours. As errors were found in models, the worst-case time spent completely regenerating and re-executing tests was under 90 minutes, but more commonly was less than 15 minutes.

Maturity was an evaluation of readiness for deployment across a potential population of several thousand engineers—e.g., if the tools crash frequently or if they produce inconsistent, incorrect, or confusing results. The study used the 9-level NASA/DoD Technology Readiness scale[3] and found the toolset to be at Level 7, "System Prototype Demonstrated in [an operational environment]". In summary, prototype software exists and all key functionality is available for demonstration or test; the tools were well integrated with operational systems; operational feasibility was demonstrated and most of the software bugs have been eliminated; and at least some documentation is available. A general deployment would require level 9 "Actual system [performance] proven through successful [developmental use]."

Conclusion

For unit test, this appears to be much more effective than the standard manual, iterative approach of writing tests, running them, checking coverage, writing more tests to fill coverage gaps, running more tests, and so on. Defining the input space to achieve required coverage consumed the largest amount of time, requiring several iterations of test case generation – especially to achieve full MC/DC. With experience, however, the number of iterations was significantly reduced. The study used staff with significant experience, but in general the approach required no knowledge or skills that could not easily be learned by an above average entry-level software engineer—e.g., creating and debugging the test generation models was much easier than writing and debugging the source code being tested.

Overall, results of the study were positive, although there are remaining issues of deployment packaging and tool licensing, training, mentoring, and technical support. Data for an empirical comparative evaluation of defect detection capability between combinatorial testing and other approaches do not exist, but there is enough evidence from literature to justify a pilot project or a trial deployment in a business unit. This is the current plan going forward.

ABOUT THE AUTHOR

Redge Bartholomew is with Rockwell Collins, currently researching tools and methods for automating the development of embedded software and for reducing the number of latent software defects found during test and evaluation.

400 Collins Road
M.S. 108-265
Cedar Rapids, Iowa 52498
Phone: 319-295-1906
E-mail: rgbartho@rockwellcollins.com

NOTES

1. The ACTS executable is available from NIST. See <http://csrc.nist.gov/groups/SNS/acts/index.html>
2. NuSMV is available from <http://nusmv.fbk.eu>
3. Technology Readiness Calculator at <https://acc.dau.mil/CommunityBrowser.aspx?id=320594&lang=en-US>

REFERENCES

1. Government Accountability Office, "F-35 Joint Strike Fighter: Current Outlook Is Improved, but Long-Term Affordability Is a Major Concern, GAO-13-309", March 2013
2. Government Accountability Office, "KC-46 Tanker Aircraft: Program Generally Stable but Improvements in Managing Schedule Are Needed, GAO-13-258", February 2013
3. Government Accountability Office, "Airborne Electronic Attack: Achieving Mission Objectives Depends on Overcoming Acquisition Challenges, GAO-12-175", March 2012
4. Jones, "Software Quality and Software Economics", SoftwareTech News, April 2010
5. Dvorak (ed.), NASA Study on Flight Software Complexity, March 2009.
6. National Academy of Sciences, Critical Code: Software Producibility for Defense, 2010
7. Baldwin, DoD Software Engineering and System Assurance, NDIA Proceedings of the 11th Annual Systems Engineering Conference, October 2008.
8. Afzal, Torkar, Feldt, Search-Based Prediction of Fault-Slip-Through in Large Software Projects, IEEE Symposium on Search Based Software Engineering, September 2010.
9. Andersin, TPI – a Model for Test Process Improvement, Seminar on Quality Models for Software Engineering, U of Helsinki, October 2004
10. Lu, Park, Seo, Zhou, Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics, ACM Proceedings of the 13th Annual International Conference on Architectural Support for Programming Languages and Operating Systems, March 2008
11. Kuhn, Wallace and Gallo, "Software Fault Interactions and Implications for Software Testing," IEEE Transactions on Software Engineering, June 2004.
12. Borazjany, Yu, Lei, Kacker, Kuhn, Combinatorial Testing of ACTS: A Case Study, Proceedings of the International Conference on Software Testing, Verification, and Validation, April 2012
13. Lei, Kacker, Kuhn, Okun, Lawrence, IPOG: A General Strategy for T-Way Software Testing, IEEE Proceedings of the Conference and Workshops on the Engineering of Computer-Based Systems, March 2007.
14. Kuhn, Lei, Kacker, "Practical Combinatorial Testing: Beyond Pairwise," IEEE IT Pro, May/June 2008.
15. Kuhn, Okun, Pseudo-Exhaustive Testing for Software, NASA/IEEE Proceedings of the 30th Software Engineering Workshop, April 2006.
16. Kuhn; Kacker; Lei, Practical Combinatorial Testing, NIST Special Publication 800-142, National Institute of Standards & Technology, October 2010.
17. Ammann, Black, Majurski, Using Model Checking to Generate Tests from Specifications, IEEE Proceedings of the 2nd International Conference on Formal Engineering Methods, December 1998.
18. RTCA, DO-178C: Software Considerations in Airborne Systems and Equipment Certification, RTCA, 2011.