Test Data Generation for Product Lines – A Mutation Testing Approach

Zoë Stephenson, Yuan Zhan, John Clark and John McDermid
Department of Computer Science, University of York

Abstract. Modern product lines typically generate large and complex software products. There is an associated cost increase from the need to test such products, especially for a safety-critical embedded system. We propose a method by which characteristics of the product line can be used as a way of reducing the test data search space and providing effective test data for relevant testing problems. We illustrate this with a solution to the problem of checking that a generated instance is a correct reflection of the required behaviour.

1 Introduction

Product lines are becoming ever more complex, and are being applied to more and more different domains as the technology begins to mature. In each of these domains, testing plays an important part. However, in the domain of embedded safety-critical control systems, testing is of vital importance to the business whereas product line technology for the domain is still relatively immature. As a broad research area, we are interested in the issue of testing methods that can help to ease acceptance of product-line technology as a means of generating engine control software.

This paper is motivated by our work with Rolls-Royce, developing product-line technique, for engine controller software. Based on our analyses to date, engine controller software has many hundreds, if not thousands, of variation points, posing a challenge for product-line testing.

2 Product Lines

In adopting product line techniques, we are interested in very simple models of product lines [Gri2000,WL99] and software architectures [Bos2000]. We make the following assumptions about the use of product line technology, as a way of scoping the problem:

- Products are derived by making a series of decisions, arranged in a tree [Par76]. Each decision is an internal tree node with at least two child nodes. The leaf nodes

represent complete products within the product line. If decisions are truly independent, then there may be multiple trees.

- The process is a simple selection among the available decisions according to a statement of the customer's requirements, shown in Figure 1. The statement of requirements is only concerned with the options and parameters that are selected in the product line, and is expected to be written in some domain-specific language so that the generation process is easily automatable. The generation of a product is expected to be significantly cheaper than the construction of individual assets or of statements of requirements. This process may involve some degree of calculation according to the needs of the domain. For example, the fault detection and accommodation logic in an engine controller may fulfil the overall input conditioning requirements by a series of comparisons and selections that among them exhibit a particular degree of tolerance for aberrations in the inputs.
- Many product lines exhibit features with markedly different options, such as entirely optional components (an example from the domain of aircraft engines is thrust reverse) or significantly different variations (pivoting-door thrust reverse as opposed to translating-door thrust reverse). For these types of variation, the selection can be readily validated against the requirements by inspecting the product. However, this type of variation is relatively rare in the engine control domain; most of the variations are controlled by wide-ranging parameters. Validating these choices by inspection can be a tedious and error-prone task.
- Test data that tests the common features of the product line is assumed to be readily available through existing test-data generation techniques. This paper assumes that the interesting technical challenges are to be found in testing for variable features, the points at which the designer makes decisions about the product to be instantiated.
- The products of interest in engine control systems will be Simulink control diagrams. These offer a very simple way of arranging components into products by instantiating elements from libraries into simulation models, which can then be simulated or automatically transformed into code.

For product-line testing in this domain, we can identify a number of relevant issues:

- Having generated a set of product-line assets, what level of testing is appropriate before their integration into a product? How much evidence from this stage of testing is admissible as certification evidence for the individual product?
- Given a product that has been generated from the product line, what test data can be found that most efficiently demonstrates that the product matches its requirements? How can this product be distinguished from different members of the same product line?

- Given an existing product from before the development of the product line, can a functionally equivalent replacement be generated from the product line?

The first issue has been extensively discussed in [McG01]. He described methods for extensive integration testing for product line assets. However we believe that this will in general be prohibitively expensive due to the large number of variation points and possible values for each variation. We suggest instead only generating tests once a configuration has been chosen. McGregor's unit level testing concept is to test all units when the product line assets are created. In our view, we are only able to perform testing at this stage for the units that do not have any internal variation points. For the remaining units, we test their behaviour once instantiated into a complete configuration. This should be more practical and cost-effective considering the potential number of variants.

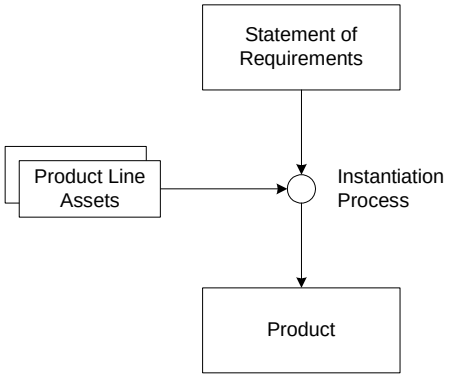

Figure 2: Generation Process

3 Testing Strategy

There are some technologies that can help us to address the other two issues. We currently have a programme of test-data generation research that is amenable to the characteristics of Simulink models. It aims to carry out testing that is effective (defects are found) and efficient (the cost of finding defects is minimised). Normally, testing is carried out so as to exercise all parts of the system under test in order to achieve high structural coverage [ZHM97]. For example, *all-statements-coverage* requires that each

statement in the system be executed at least once by one of the test inputs. This helps to increase confidence in the system under test.

Simply executing a piece of code or a part of system cannot ensure that errors are exposed. As an example, if there is a statement in the system under test that is wrongly defined as `y = x × 5` instead of `y = x × 15` (which it should be), a test input that renders a runtime value of `0` for `x` will fail to detect this error. Therefore, the concept of mutation testing is introduced [VM97]. Mutation testing focuses on the generation of test data with the ability to detect particular faults (i.e. being able to demonstrate an error in the outputs). Given two models with only minor variations between them, one being the correct model and the other being an erroneous model, we have implemented a tool that can automatically generate test data to distinguish between those models. Such automation is based on the ability of simulating or running models. The detailed application of the technology can be found in [YC04]. It is well known that test-data generation is one of the most tedious and costly processes within software testing. The automation of test-data generation would naturally reduce the cost of testing. Our interest here is in how to ensure efficient and effective test generation for product lines.

4 Proposed Approach

Our test-data generation technique can find test inputs to ensure that small variations between two models can be detected by examining the outputs. We can also apply this technique to test data generation for the purpose of distinguishing slightly different product line instantiations. Consider the problem of determining whether an instantiation of the product line matches the requirements. Suppose there is a decision between two options, A and B. The set of products in the product line can be partitioned according to whether option A is chosen, or option B. Then, test data can be generated such that the results of the tests are sufficient to show the difference between systems with option A and systems with option B. The output of such tests can be compared with the requirements to show that the desired feature is the one that has been implemented.

This test-data generation technique relies on the execution or simulation of systems and therefore generally requires complete instantiated products or components in order to generate successful test data. In some cases it may be possible to choose an arbitrary instantiation of the remaining features, but there are some characteristics of variant products that mean this is not a general solution:

- The technique works by comparing the output values for a given set of input data applied to different variants. It may be the case that for some instantiations of remaining features, a change can be detected in the outputs, but for other instantiations of remaining features, the change is masked.
- There may be dependent features that depend on the choice being tested. If so, then it will not be possible to keep all of the remaining features the same on both sides of

the decision being tested. This will make it difficult to separate out output changes that are inherent in the decision being tested from output changes that are relevant to the dependent features.

- A typical product line will have parameters that cover a far larger decision space than just simple options. Even a set of decisions with only a few options each will quickly cover an enormous space of possibilities. Checking each instance to show that the test data demonstrates a difference in one feature will not be a practical solution.

To address these concerns, we take a simple staged strategy:

- Features must be independently distinguished by output values. Some features will be independently distinguished by outputs from the system as a whole; for other features, the output values must come from internal connections within the system. We can ensure that this is possible by inserting probes into the system under test and recording the values at those points.
- There are often more than two options for a given decision. In these situations, if there are a small number of options, then the option to be distinguished is tested against all the other options. Otherwise, if there are a large number of options (such as with integer parameters) a sampling method is used instead. It is assumed that a random sample will produce an adequate result here, although directed search techniques can be applied if needed.
- A given feature may affect more than one component, such as a matched set of input and output components to handle a particular type of control device. It may be the case that only some of the required components have been put into place, or that some of those components have an incorrect configuration for the given variant. To test for these situations, the test data generation process must be applied recursively to the components of a feature, to find test data that causes a different output under the different configurations of those components.

This provides a set of test data that will distinguish between the option that is expected to be chosen, and other options. This test data can then be checked against the statement of requirements for the feature in question, to show whether the functionality provided by the product for this particular feature represents the desired behaviour. This type of feature-directed testing has the following additional benefits:

- The testing problem is reduced from testing the whole product to testing for variant features. This reduces the space in which to test, improving testing effectiveness, at the cost of only being able to test variant features. However, it is expected that the testing of non-variant features will occur far less frequently than that of variant features, and so there is a business case for the use of the strategy.
- The test sets that are created will be matched to specific outputs from the product, reducing the set of outputs that must be checked against the requirements to demonstrate that a feature is correctly implemented, and guiding the test engineer to focus on those outputs. Other output values may still be relevant, but the priority is given to those that are determined by the test data generation process.

5 Future

From this discussion, we can identify some further issues of interest:

- Are the assumptions realistic? Will it be possible to isolate common features and provide test sets that can be reused?
- Are the assumptions relevant to other domains? Will the strategy outlined here be applicable to other types of system?
- How adequate will these tests be in general? Will they only be usable as confidence-building during product instantiation and rework, or will they be admissible as part of a safety argument?
- What kind of coverage is produced by tests such as these? Is there a concept of variation coverage that must be addressed? Should a variation coverage metric be investigated?

These issues are expected to be addressed through experimental work in testing product line products.

References

1. Yuan Zhan and John Clark. Automatic Test-Data Generation for Testing Simulink Models. Technical Report of University of York.
2. John D. McGregor. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022. http://www.sei.cmu.edu/publications/documents/01.reports/01tr022.html
3. Jeffrey Voas and Gary McGraw: Software Fault Injection: Innoculating Programs Against Errors. By John Wiley & Sons, 1997.
4. Hong Zhu, Patrick A. V. Hall and John H. R. May: Software Unit Test Coverage and Adequacy. ACM Computing Surveys, Vol. 29, No. 4 December 1997.
5. D. M. Weiss and C. T. R. Lai. Software Product-Line Engineering: A Family-Based Development Process. Addison-Wesley, 1999.
6. J. Bosch. Design and Use of Software Architectures. Addison-Wesley, 2000.
7. M. L. Griss. Implementing Product-Line Features with Component Reuse. In Software Reuse: Advances in Software Reusability – Proceedings of the Fifth International Conference on Software Reuse. pp. 137-152, June 2000.
8. D. Parnas. On the Design and Development of Program Families. IEEE Transactions on Software Engineering, 2(1) pp. 1-9, March 1976.