JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 2, March-April 2009

Variation Verification

John D. McGregor, Clemson University and Luminary Software LLC, U.S.A.

Abstract

The asset base of a software product line organization includes many types of assets. The thing that binds them together is the range of variation they must accommodate. These differences make verification even more difficult than usual. In this issue of Strategic Software Engineering I will discuss the relationship of verification to variation management.

1 INTRODUCTION

Verification is an important part of any product development effort. Determining that the product satisfies its requirements is important in any market but in life-critical systems it is a legal requirement. Software product line organizations often have a goal of higher quality whether their products are safety-critical or not. In my opinion, the single most strategic mistake that organizations make in the early stages of software product line adoption is to limit verification activities to only software modules.

There is a large base of literature on verification techniques. Much of it is limited to software but there are inspection and review techniques and other types of activities for specific situations such as documents or models. I don't intend to propose yet another verification technique. What I do propose to do is consider how the verification process might be expanded to accommodate the range of variation required for the scope of products in the product line and the range of assets constructed by the product line organization.

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market or mission, and that are developed from a common set of core assets in a prescribed way, according to the definition used by the Software Engineering Institute (SEI) [Clements01]. The list of assets is long and varied: software architecture, tests, business cases, and more. In a perfect world there would be a verification activity for each asset, but in reality we may have to settle for building reusable verification assets that can be easily, perhaps even automatically, applied repeatedly as the asset base of the product line evolves.

Each product in the product line is a complete product and has its own set of requirements, tests, user's manual, and justification. The goal of the product line organization is to produce each product from reusable assets. Those assets are built with the anticipation that in order for them to be used in multiple products they would have to be configured for each specific product. The configuration is accomplished by building the assets as reusable “templates.” Each template is instantiated for use via some mechanism. Verification of the templates, instead of the instantiations, is key to successful verification in the product line.

A central element in a software product line is the variation between products. Obviously the more variation, the less reuse and the less profit, but that is not my concern at the moment. My interest is in how the variation, however much there is, is managed and exploited. First I want to examine strategies for managing variations and then I will relate this to the verification efforts.

2 PRODUCT LINE VARIATION

In a software product line the products differ from each other. In addition to constructing multiple product specifications, the product line organization needs to identify how the products differ. The analysis of commonalities and variabilities often begins with a feature model to capture the similarities and differences between products in the product line. A feature is any user-visible attribute of a product. It may be a capability, such as being able to show .wmv formatted files, or it may be a quality, such as secure data storage. This model provides a basis for leveraging the commonalities and managing the variabilities among the products.

Feature models have a syntax that allows the identification of features that must be in every product, features that may be selected or not, and groups of features from which a subset may be selected for a product. Those features that are not in every product lead to the notion of variations. Some variations are directly related to the goals of the product line and some are related to how those variations are realized in the products. We call them strategic and tactical variations respectively.

Strategic variations are those differences that are the result of market or technology choices. For example, certain safety features are required in one country and not in another. If the company wishes to sell the vehicle in both countries either they include the safety equipment on both models or this is a point at which the products vary. This is the starting point of a verification thread. Strategic variations are the result of roadmapping the markets and technologies and identifying products [Petrick 05].

Tactical variations are different ways of realizing the same strategic variant. The company might purchase the special safety equipment from companies close to each assembly plant. This is a traceability mechanism that bridges between the strategic variations and where they are realized in individual assets [Chastek 09]. This bridge is missing in many product line organizations. They go directly from business goals to variation points with the result that changes to the goals are difficult to propagate to individual assets.

The strategic and tactical variations lead to specific points in the product line assets that must be different from one product to another. These variation points are represented in the products by specific variation mechanisms. For example, a Word document might have conditional text that is included in a document or not depending on a parameter provided to the document. A makefile may decide which modules to link based on some environment variable or configuration file.

Feature modeling is limited to product features, but the differences affect more than just the software. Standard variation models capture constraints among the features defined in the feature model. The feature model and the associated constraints must be linked into the greater asset base. I will define a model that captures the variation points and their relationships among variation points in all of the assets.

For example, a strategic variation is related to multiple tactical variation definitions. Each tactical variation is related to variation points in multiple assets. A software feature can be traced from the strategic level through the tactical variations to the individual variation points in such assets as the test plans, the architecture, and the software.

3 PRODUCT LINE ASSURANCE CASES

Safety-critical systems that require approval from regulatory agencies, such as the FAA and the FDA in the United States, must present evidence of the quality of their product and the process by which it was built. An "assurance case" provides an organizing device for the systematic argument that is presented to support a claim of sufficient quality [SEI 09]. The strategic variation points of a product can help organize an assurance case because they are directly related to the goals of the product line which includes the quality attributes required for the products to succeed.

The basic structure of the argument is a series of: "claim of the absence of a certain type of risk due to the development method" directly linked to "evidence that supports the claim". Various types of verification activities contribute to the body of evidence. Testing, reviews, architecture evaluations, process audits all contribute information. The goal is to at least reuse verification activities across products in the product line but even more, the reuse of verification results. That is, we would like to verify an asset and then reuse it without re-verifying. The techniques discussed in section 3 address producing results and designing the system so that verification results can be reused.

Even unregulated organizations can use the assurance case approach to unify the verification activities in their organization. Verification is usually spread over the full life cycle and across functional groups. If there is a verification group it is often outside the development group and not as closely involved in the effort. The assurance case is based on the qualities desired in the final product not the sequence of process tasks. This requires that the organization gather and synthesize actions taken at various times during development and describe their impact on a specific quality and to do this for each quality that is important to them.

Assurance cases can be constructed with variation points just like any asset. Then product artifacts that are shared across products can be addressed once and individual variants can each also be addressed once. The assurance case for a specific product is assembled from the assurance case templates and choices at each variation point. Evidence is added for any product-specific features.

4 VARIABLE VERIFICATION

In the development of a one-off product, verification is not easy particularly when the requirements are vague or changing or both. In a product line organization, where the development of multiple products is being managed, this is further complicated by the desire to reuse the information created to verify one product for the verification of other products. There are a number of issues with this type of reuse and how we overcome the problems that arise.

How can the contextual differences among the products be determined? The context is the environment surrounding the product. We need to know the differences so that we can infer how an asset that works in one product might need to be changed to work in another product. This identifies possible mismatches that could occur between the verification assets and the environment in which they would have to work. What constraints and assumptions exist in one product that do not in another? For example, one product may be different from another in the level of security required. Scenarios used to verify the one product must include security actions while the scenarios for the other can not.

The variation point definition should be thought of as a specification for the allowable variants. The information in that definition must put sufficient constraints in place to communicate to product builders and other core asset builders. The specification should guide the development of variants by stating which attributes are important and what are the minimum levels required. There are a number of formal constraint notations that can be used but most do not provide the expressiveness needed.

How can the dependencies among assets be resolved? If design firewalls such as interfaces are not used, the software becomes monolithic and reuse becomes cumbersome. If you need one specific module you may have to take a cluster of modules due to dependencies. The supporting assets need to be as modular as the software is. The modularity should be at the same grain size in both the software and non-software assets so that the variants in the product implementation can be readily mapped to the variants in the verification assets. This allows the verification assets to be instantiated in coordination with product instantiation.

The use of appropriate variation mechanisms will assist in the design of an asset. I have previously described how to use XML-based Variant Control Language (XVCL) to construct documents with variation points [Zhang 04]. Code, Word documents, most any format can be structured using the frame-based approach of XVCL. Variant values are passed to the XVCL engine and used to control the order in which information is combined. Tests sets are implemented as either software or at least scripts. The Eclipse TPTP, including JUnit, uses these techniques.

How can the relationships among the pieces be managed? The variants are related to the asset to which they belong but they are also related to the assets that will be used to verify the product and the assets used to document the assets. These variants may be related statically or dynamically. The verification assets may directly reference the product assets, but it would be poor design to have the product assets reference the verification assets. Using meta information, a configuration management system can be used to link the appropriate versions of verification, documentation, and product assets.

One technique for managing these relationships is to embed the relationships in a domain specific language (DSL). The DSL defines terms that can have much more specific constraints than generic design constructs in UML or SysML. This automatically reduces errors since certain errors simply can not be expressed in the language. The DSL should be useful to express the constructs needed in the product and in the verification of the product. [Chandra 99] provides an example of how to use the DSL for verification. [Völter 09] provides an interesting example of the definition of a DSL.

False positives

The verification activities indicate that the asset is correct when it is not. This can be a dangerous mistake in safety critical system, but we can’t double check every positive result. One of the ways to obtain a false positive is to have a less than complete test set. My previous discussions of Orthogonal Array Testing [McGregor 01] indicated a way of knowing what percentage of possible tests were actually being constructed and run. In particular, the OATS approach can accommodate the multiple variants that may be substituted at a particular variation point. Other combinatorial test specification techniques can also give estimates of the percentages of possible tests that are actually being created.

False negatives

The verification activities indicate that the asset is not correct when it is. This is safer than a false positive but still expensive. Work is done unnecessarily to attempt to fix what isn’t broken. One particular cause of false negatives is incorrect tests. The Guided Inspection technique, which I discussed previously [McGregor 98], can be used to verify the correctness of tests. Guided Inspection is effective because it is a review process guided by test cases. The test cases are scenarios that are derived directly from the use cases of the system under development. In a product line environment the Guided Inspection scenarios can be reused with various configurations of the assets.

5 AN APPROACH

Much of what I have discussed so far is compatible with a model driven development (MDD) approach. I want to explore further the implications of doing verification in an MDD organization. Basically the guidelines I give here correspond to the reasons why modeling is useful.

Associate verification activities with specific product artifacts. Although the verification activities are often performed by a group separate from the developers, the verification artifacts should be associated with the product artifacts they are used to verify. One advantage of modeling is the ability to maintain multiple views of the model. Entities can show up in multiple diagrams and when one representation of the artifact is changed all others are automatically updated. Associating artifacts maintains the congruency between the artifacts. For example, in MDD the output of one tool is often transformed into the input for another tool by performing a transformation on the output. The transform is itself an asset that should be verified before the transformed data is assumed to be correct [QVT 09]. Associating a verification process and test data with the transformation makes it easy to re-verify the transform when a portion of it must be changed due to product asset evolution.

To make these associations useful, you should …

Model the verification artifacts just as you would product artifacts. Once verification artifacts are in the model and associated with the corresponding product assets, constraints between the two maintain the correspondence. One of the main reasons for modeling product artifacts is to save time and money. Since verification costs are a large percentage of development costs, not modeling the verification assets misses a major opportunity. The savings doesn’t come in the first round of model building. The true benefit is the reduction in the time required to modify the artifacts to match changes in product artifacts. The reduction is partially due to the improvement in traceability and due to the automatic updating triggered by the constraints. In a product line you not only get reuse over the evolution of a single product, you get reuse over the set of products for most of the artifacts.

Of course, to take advantage of these artifacts across time you must …

Maintain all the models. The expense of maintaining models is in direct proportion to their usefulness. Drawing pretty, or not so pretty, pictures using a drawing program does not begin to tap the power of the models. Software engineering environments such as Eclipse [Eclipse 09], Topcased [Topcased 09], and OSATE [OSATE 09] provide many tools that are context sensitive and, to a limited extent, understand the models being built. These tools provide a first line of error checking by restricting the statements that can be written to syntactically correct statements. The tools can be used to analyze models for a variety of attributes and metrics. They also have facilities to automatically generate some or all of the representation that is the ultimate goal of the model.

The power of these models is only realized if you…

Always verify the verification artifacts. Another reason for modeling is to improve the correctness of the final product. False results, both positive and negative, from tests can be the result of incorrect or incomplete tests and requirements. Verifying the verification artifacts is just as important as verifying the product. In a product line context the importance is magnified by the number of products for which the verification artifacts will be used.

6 SUMMARY

Verification in a software product line organization requires a thorough understanding of the variations among the products. This understanding will evolve as the product line scope evolves. An effective way to manage that is to model the variations and then maintain the model. Model driven development techniques incorporate this seamlessly into the portfolio of models.

The strategic goals of most product line organizations include improved quality and maintainability. A systematic verification process helps achieve those goals by ensuring that verification activities are cost effective and thorough. A comprehensive variation model makes a strategically significant contribution to the success of the verification of the product line.

REFERENCES

[AADL 08] Architecture Analysis and Design Language, http://www.aadl.info/, 2008.

[Chastek 09] Gary J. Chastek and John D. McGregor. Modeling Variation in Production Planning Artifacts, VaMoS 2009.

[Chandra 99] Satish Chandra, Bradley Richards, and James R. Larus. Teapot: A Domain-Specific Language for Writing Cache Coherence Protocols, Microsoft, http://research.microsoft.com/apps/pubs/default.aspx?id=72011, 1999.

[Clements 01] Paul Clements and Linda Northrop, *Software Product Lines: Practices and Patterns*, Addison Wesley, Reading, MA (2001).

[Eclipse 09] Eclipse Foundation, www.eclipse.org.

[McGregor 08] John D. McGregor. Mix and Match, Vol. 7, No. 6, July-August 2008, http://www.jot.fm/issues/issue_2008_07/column1/

[McGregor 01] John D. McGregor and David A. Sykes. A Practical Guide to Testing Object-oriented Software, Addison-Wesley, 2001.

[McGregor 98] John D. McGregor. *The Fifty Foot Look at Analysis and Design Models*, Journal of Object-Oriented Programming, July - August 1998.

[OSATE 09] www.aadl.info.

[Petrick 05] Irene J. Petrick. Roadmapping as a mitigator of uncertainty in strategic technology choice, *Int. J. Technology Intelligence and Planning*, Vol. 1, No. 2, 2005.

[QVT 09] QVT, http://www.omg.org/docs/formal/08-04-03.pdf, 2009.

[Rech 08] Jorg Rech, Christian Bunse. Model-Driven Software Development: Integrating Quality Assurance, Idea Group Inc (IGI), 2008.

[SEI 09] Software Engineering Institute, http://www.sei.cmu.edu/pcs/acprep.html, 2009.

[SYSML 09] SysML, www.omg.org, 2009.

[Topcased 09] Topcased, http://www.topcased.org, 2009.

[UML 09] Unified Modeling Language, http://www.omg.org, 2009.

[Völter 09] Markus Völter. Architecture as Language: A story. http://www.infoq.com/articles/architecture-as-language-a-story.

[Zhang 04] Hongyu Zhang and Stan Jarzabek. XVCL: a mechanism for handling variants in software product lines. Science of Computer Programming, 53(3), 381 – 407, 2004.

About the author

Dr. John D. McGregor is an associate professor of computer science at Clemson University, a visiting scientist at the Software Engineering Institute, and a partner in Luminary Software, a software engineering consulting firm. His research interests include software product lines and component-base software engineering. His latest book is *A Practical Guide to Testing Object-Oriented Software* (Addison-Wesley 2001). Contact him at johnmc@lumsoft.com.