A Pattern-based Modeling Approach for Software Product Line Engineering

Julie Street Fant
The Aerospace Corporation
julie.fant@aero.org

Hassan Gomaa
George Mason University
hgomaa@gmu.edu

Robert G Pettit
The Aerospace Corporation
robert.pettit@aero.org

Abstract

This paper addresses the problem of variability in software product line (SPL) engineering by first considering different SPL model-based approaches and in particular the tradeoff between the development effort in domain engineering vs. application engineering. The paper then describes a SPL engineering approach, which is both model-based, and pattern-based. This approach bases the SPL software architecture on software architectural patterns and relates these patterns to SPL features. The paper describes how this approach has been applied to an unmanned space flight SPL.

Keywords: Software product lines, UML, software architectural design patterns, domain engineering, application engineering, space flight software.

1 Introduction

A software product line (SPL) consists of a family of software systems that have some common functionality and some variable functionality [1, 2, 3, 4]. Software product line engineering (SPLE) involves domain engineering, in which the requirements, architecture, and components for a family of systems are developed, and application engineering, in which products are developed and configured from the product family assets.

The Object Management Group (OMG) advocates model-driven architecture in which “modeling is designing of software applications before coding”. Modern software modeling approaches [3], provide greater insights into managing commonality and variability by modeling SPLs from different perspectives, which are referred to as software modeling views. This paper uses a UML model-based approach for developing multiple views of a SPL based on the PLUS method [3].

Software architectural patterns [3, 10] provide the template for the overall software architecture of an application. Basing the software architecture of a SPL on one or more software architectural patterns helps in designing the software architecture. However, to apply patterns effectively in SPLE, it is necessary to incorporate variability into the software patterns. In a feature-based SPLE approach, it is also necessary to relate features to patterns, as described in this paper. This paper describes how the approach has been applied to an unmanned space flight software (FSW) SPL.

This paper is organized as follows. Section 2 describes related work; Section 3 describes and compares three different model-based approaches to SPL engineering; section 4 describes the FSW SPL case study. Next, the architectural design pattern modeling approach for SPL is described in Section 5 and application derivation for the FSW SPL is described in Section 6. Finally, there is a description of how this approach is validated in Section 7 and a discussion in Section 8.

2 Related Work

There are many notable SPL engineering approaches including [1–4]. An important part of developing a SPL is separating the common and variable parts of the SPL, typically referred to as commonality/variability analysis [4]. Feature modeling [12] is crucial for managing variability and product derivation as it describes the SPL requirements in terms of commonality and variability, as well as defining the SPL dependencies [3]. Many SPL approaches focus on capturing all possible variability in the domain engineering phase. Other approaches focus on modeling the commonality of the SPL and define variation points where variability is permitted to be introduced during application engineering [5]. However, this approach defers most development of variable components to application engineering. The approach described in this paper takes a different approach by modeling SPL variability at the architectural design pattern level.

Other related work includes approaches to build software architectures from design patterns [3, 6–8] and software architectural design patterns [9–11]. These approaches do not explicitly capture variability in pattern selection for SPL members and do not address variability within design patterns.
This paper builds on previous work in [3,14–16]. In [14] executable architectural design patterns are applied to FSW architectures. This paper extends this work by defining a SPL approach that addresses variability within design patterns. In [15] the concept of using a feature to design pattern mapping was briefly introduced at a high level. In [16], an overview of the approach is given. This paper extends [15] to capture variability in design pattern selection for SPL members and extends [16] to provide a more detailed description of the approach and feature to design pattern mapping.

### 3 Model-based SPLE Approaches

In model-based software design and development, models are built and analyzed prior to the implementation of the system, and are used to direct the subsequent implementation. A better understanding of a system or SPL can be obtained by considering the multiple views [3, 13, 19], such as requirements models, static models, and dynamic models of the system or SPL. A graphical modeling language such as UML helps in developing, understanding and communicating the different views. A key view in the multiple views of a SPL is the feature modeling view [1, 12]. To enable the integration of feature modeling into this multiple view model-based approach, feature modeling also needs to use the UML notation as described in the PLUS method [3].

#### 3.1 Modeling Variability in SPL

This section describes three different model-based approaches for SPL engineering, which address the tradeoff between the development effort in domain engineering vs. application engineering. Many SPL approaches [1, 2, 3, 4] advocate that domain engineering encompasses the development of all variable components in addition to all common components. One such approach is the Evolutionary Software Product Line Engineering Process (ESPLEP) used by the PLUS method as described in Section 3.2.

A concern expressed by some potential adopters of SPLE is the significant upfront investment in domain engineering before the first application is derived. There may also be situations where it is not possible to develop all the variable components because not all the SPL variability is known at domain engineering. Furthermore, there may be cases where the variability in an application is one of a kind, in which case, it might be considered preferable to develop the variable application components during application engineering rather than during domain engineering. One approach to allay these concerns is to only develop the common components, referred to as the common core, during domain engineering. Details are then provided on how to introduce variability into the core, so that variable components can be developed during application engineering. One such approach, called the Variation Point Modeling (VPM) Approach, is briefly described in Section 3.3.

A third approach described in this paper lies in between ESPLEP (in which variable components are developed during domain engineering) and VPM (in which variable components are developed during application engineering). This third approach addresses the problem of variability in domain software architectures by specifying this variability at a higher level of granularity through architectural design patterns. Several different combinations of specific components, connectors, and interactions are abstracted into one design pattern. This approach requires less modeling and design during domain engineering. The tradeoff is that the application engineering process necessitates additional modeling and design since the individual application specific components and interactions must be developed from the design patterns.

#### 3.2 Evolutionary SPL Engineering

The Evolutionary Software Product Line Engineering Process [3] is a highly iterative software process that eliminates the traditional distinction between software development and maintenance.

![Figure 1. Evolutionary SPL process model](image)

Furthermore, because new software systems are outgrowths of existing ones, the process takes a SPL perspective; it consists of two main processes, as depicted in Fig. 1: a) Domain Engineering. A SPL multiple-view model, which addresses the different views of a SPL, is developed. The SPL multiple-view model, SPL architecture, and reusable components are developed and stored in the SPL reuse library. b) Application Engineering. The user selects the required features for the individual SPL member. Given the features, the SPL model and architecture are adapted and tailored to derive the application architecture.
The kernel software architecture represents the commonality of the SPL. Evolution is built into the software development approach because the variability in the software architecture is developed by considering the impact of each variable feature on the software architecture and evolving the architecture to address the feature. Evolutionary software development allows the SPL developer to decide how much of the variable parts to develop during domain engineering ranging from none of the variable parts to all of the variable parts. In [3], the approach used is to describe developing the commonality initially followed by all the variability.

3.3 Variation Point Model

In domains where all the variability is not known at initial domain engineering or the investment in developing all the SPL assets upfront is considered too high, the Variation Point Model (VPM) [5] is an interesting solution. With this approach, the common assets are developed during domain engineering together with detailed instructions as to where and how the domain architecture can be extended to introduce the variability needed for a given application. Modeling variability using variation points exposes the variation points to the reuser. This approach allows reusers to create their unique variants and maintain them. If the number of applications is limited, then the added flexibility is not as relevant. However, if the SPL is a complex one where the core asset developer could not possibly create all the variants, then modeling variability with variation points is advantageous. It is critical for the reuser to understand the variation points and how to use them to create variants. The VPM modeling approach targets this situation.

4 Space Flight Software Case Study

The Space Flight Software (FSW) SPL is used to illustrate the pattern based SPL approach. The FSW SPL involves controlling unmanned spacecraft to achieve its mission. The FSW domain was selected because the capabilities of the FSW SPL members can vary significantly. For instance, a spacecraft may rely extensively on the ground station to control the spacecraft and may require a small amount of hardware to perform its mission. Another spacecraft may utilize onboard autonomy to control the spacecraft and may have extensive hardware to perform its mission. The functionality of other spacecraft can vary anywhere in between these extremes. Thus the FSW SPL is an ideal domain to apply this pattern-based design approach because of its architectural variability. This paper describes the application to the FSW SPL’s command and data handling subsystem.

We derived two SPL members from the FSW SPL. The two case studies are real-world programs that cover a wide variety of functionality, thus many different SPL features are exercised. The first case study was Student Nitric Oxide Explorer (SNOE). SNOE [18] is a small satellite whose mission involves using a spin stabilized spacecraft in a low earth orbit to measure thermospheric nitric oxide (NO) and its variability. The spacecraft contains four payload instruments, which are an ultraviolet spectrometer, an auroral photometer, a solar soft X-ray photometer, and a microGPS Bit-Grabber Space Receiver. SNOE is a low earth orbiting satellite that relies heavily on the ground station to control the spacecraft’s small amount of hardware and payloads.

The second case study was Solar Terrestrial Relations Observatory (STEREO). STEREO [21] contains two nearly identical large three-axis stabilized spacecraft in heliocentric orbit, which is an orbit around the sun. Since the spacecraft is far away from Earth, the STEREO FSW contains more autonomous functionality and relies less on real-time ground commanding. Additionally, it requires guidance and control algorithms along with propulsion hardware to achieve and maintain its orbit. The FSW also needs to support maneuvering for its scientific data collection, communications, power generation, and thermal control.

5 Domain Engineering with Architectural Design Patterns

The design pattern approach to SPL engineering involves creating domain specific patterns, which are based on well-known general purpose patterns[13, 14], during the domain engineering phase. The purpose of creating SPL architectural design patterns is to add SPL domain knowledge and variability to the general purpose architectural design patterns, as described next. The architectural design patterns need to be related to the feature model in order to allow the selection of the appropriate design patterns for application engineering. The main steps are: 1) Build a set of variable distributed real-time and embedded (DRE) architectural design patterns that can be leveraged as a starting point, 2) Develop a set of use cases and features that are
used to define the SPL, 3) Create a feature to design pattern mapping, 4) Customize the variable DRE design patterns to become variable domain specific design patterns. The subsections below describes each of the main steps in more detail and describe the application to the FSW SPL’s command and data handling (C&DH) subsystem.

5.1 Variable DRE Design Patterns

The first step in our approach is to create variable DRE architectural and executable design patterns. These variable design patterns serve as the foundation for SPLs built using our approach. The purpose of the variable DRE architectural design patterns is to specify the variable components in the design pattern. Each variable DRE architectural pattern contains several UML views including collaboration diagrams, interaction diagrams, and component diagrams. Variable design patterns are modeled at the DRE level so that they can be reused and customized across multiple domains. Each variable DRE design pattern is supplemented by an executable design pattern that consists of interacting objects that execute state machines. The purpose of the executable version of the design pattern is to specify the internal behavior of a representative set of the pattern’s objects and to facilitate validation of the pattern. Each executable design pattern is individually simulated and validated using Harel’s approach of executable object modeling with statecharts [17]. We created 21 DRE design patterns [14] using this approach. The approach enables architectures produced by interconnecting these design patterns to be fully executable and validated, as described in Section 7.

5.2 Use Case and Feature Modeling

Next, SPL use cases are developed applying the PLUS method [3]. This involves identifying the use cases and variability within the use cases through variation points, optional and alternative use cases. From the use case variability, an initial feature model is developed [3]. Features, which represent common and variable characteristics or requirements of the SPL, are analyzed and categorized as common, optional, or alternative. Related features can be grouped into feature groups, which constrain how features are used by a SPL member.

When use case modeling was applied to the FSW SPL C&DH subsystem, three kernel use cases with numerous internal variation points were identified. These variation points resulted in identifying 52 features. A subset of the FSW SPL’s feature model is shown in Fig. 2, in which there is an <<exactly-one-of feature group>> called Command Execution feature group that has three <<alternative>> features. The Low Volume Command Execution feature is used when a small amount of commands needs processing, the High Volume Command Execution feature is used when a large amount of commands needs processing, and Time Triggered Command Execution is used when commands must be executed with strict temporal predictability. There is also a significant amount of variability in the amount and type of hardware that must be commanded, which are captured in variation points and modeled as optional features. The optional Heater feature determines whether or not the FSW is required to send commands to a heater. The antenna feature group allows one or more antennas to be selected for the FSW application.

5.3 Feature to Design Pattern Mapping

The next step is to create a feature to design pattern mapping. The purpose of the feature to design pattern mapping is to determine which variable design patterns could be mapped to SPL features. To accomplish this goal, a dynamic SPL interaction model [3] is created for each feature, which captures the objects and object interactions that realize each feature. Then the dynamic interaction models are analyzed to identify where variable design patterns can be applied in the SPL and then relates these patterns back to the SPL features. Features that are mapped to variable design patterns are called pattern specific features. Pattern specific features are coarse grained features that relate to a design pattern and differentiate among other related features. Pattern variability features are fine grained features, which influence the variability within a pattern specific feature.

The feature to design pattern mapping is demonstrated using the Low Volume Command Execution feature. The interaction model for the Low Volume Command Execution alternative feature is shown in Fig. 4. Since this feature is typically associated with small spacecraft, only the kernel input, output, and IO devices are modeled. The interaction model reveals that due to the small amount of commands that need processing, one controller could receive a set of ground commands, determine the appropriate sequence of commands and execute the commands by invoking the appropriate actions on the input, output, and IO objects. The objects and interaction sequence supporting this feature are consistent with the Centralized Control design pattern [13,14]. Thus the Low Volume Command Execution feature is categorized as a pattern specific feature and is mapped to the Centralized Control design pattern. When applied to the FSW SPL, a SPL interaction model was created for each of the pattern specific features, 24 in total. A subset of the FSW C&DH feature to design pattern mapping is shown in Table 1.
5.4 SPL Architectural Design Patterns

The next step is to derive the variable SPL architectural and executable design patterns from the variable DRE architectural and executable design patterns for each of the pattern specific features.

![Figure 2. Subset of FSW SPL feature model](image)

**Table 1. Subset FSW SPL C&DH Feature to Design Pattern Mapping**

<table>
<thead>
<tr>
<th>Feature Group</th>
<th>Va.</th>
<th>Feature</th>
<th>DRE Design Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;exactly-one-of feature group&gt;&gt; Command Execution</td>
<td>alt.</td>
<td>High Vol. Command Execution</td>
<td>Hierarchical Control</td>
</tr>
<tr>
<td></td>
<td>alt.</td>
<td>Low Vol. Command Execution</td>
<td>Centralized Control</td>
</tr>
<tr>
<td></td>
<td>alt.</td>
<td>Time-Triggered Command Execution</td>
<td>Distributed Control</td>
</tr>
<tr>
<td></td>
<td>alt.</td>
<td>High Vol. Command Execution with Command Flexibility</td>
<td>Hierarchical Control with Command Dispatcher</td>
</tr>
<tr>
<td></td>
<td>alt.</td>
<td>Low Vol. Command Execution with Command Flexibility</td>
<td>Centralized Control with Command Dispatcher</td>
</tr>
<tr>
<td></td>
<td>alt.</td>
<td>Time-Triggered Command Execution with Command Flexibility</td>
<td>Distributed Control with Command Dispatcher</td>
</tr>
</tbody>
</table>

![Figure 3. FSW SPL Centralized Control collaboration diagram](image)
The purpose of the variable SPL design patterns is to add domain specific knowledge to the design patterns so they can be systematically incorporated into SPL architectures. This process involves updating DRE pattern components, interactions, and component behavior to be SPL specific.

First, the general purpose components in the architectural design patterns are updated to be SPL specific. This includes identifying SPL components, their multiplicities, and variability. This process is illustrated using the Centralized Control design pattern in the FSW SPL (Fig. 3), which is a good pattern for handling a low volume of ground commands. This design pattern captures the I/O devices that the FSW interacts with, based on the ground commands it receives. To realize this mapping, a general purpose Centralized Control design pattern [14], which consists of a control component controlling several input and output components, is updated to give the CDH Centralized Controller and FSW I/O components in Fig. 3. SPL components are categorized as kernel, optional, or variant. For example, the SPL requires ground commands that adjust the spacecraft’s attitude by invoking actions on attitude control devices. This is achieved by providing an output component called Attitude_Control_Device_OC, which is a kernel component, because it is required in all SPL members. Since there can be different versions of attitude control in different missions, different attitude control devices are modeled as variants to the Attitude_Control_Device_OC. Additionally, since a heating device is optional, a Heater_OC output component is modeled as an optional component. While this pattern may look overly centralized, the Centralized Control design pattern is only used when there are a small amount of commands to execute and hardware to control. When this pattern is customized to an application, only an application specific subset of the optional devices is selected.

For each design pattern, an interaction diagram must be developed to reflect the needs of the SPL. If the precise sequence of object interactions is known, then it should be modeled. However, in design patterns where there is high amount of variability in the object interactions, then only a representative set of object interactions is modeled. In that case, detailed interaction modeling is deferred to the application engineering phase. For example, Fig. 4 shows an interaction diagram for the Low Volume Command Execution pattern. This pattern captures the FSW processing and execution for a set of ground commands, which involves the CDH Controller invoking actions on the input, output, and IO components. The type and amount of input, output, and IO components in the FSW Centralized Control design pattern is influenced by the variability in the SPL. For example, the Heater component is optional because not all spacecraft have heaters. The specific Heater variant is not modeled until the application engineering phase. Similarly, a variant Antenna component is depicted in Fig. 4, although the specific antennas are also not modeled until application engineering. Due to the amount of variability in this scenario, only a representative set of object interactions was created for the Low Volume Command Execution pattern, as depicted in Fig. 4.

5.5 Executable Design Patterns

The next step is to create an executable version of each design pattern for the SPL. This involves a SPL specific state machine for each component, which consists of states, events, and actions. Each executable design pattern is individually simulated and validated using executable object modeling with statecharts as described in [14].

This process is illustrated using the state machine for the normal mode of the state dependent CDH Centralized Controller (Fig. 5). The states correspond to the major CDH states of validating a command, executing the command and logging the command, with rejecting the command as an additional state. The state machine returns to idle state after completing the command processing. If a component needs to send a message to another component, then this is modeled as an action. Other application specific logic, such as a processing algorithm, can be modeled as activities.

6 Application Engineering with Architectural Design Patterns

After the development of the FSW SPL architecture and components, applications were derived from them. This is accomplished by first selecting the FSW SPL features based on application’s requirements. After the features are selected, the appropriate FSW SPL executable design patterns
are determined from the feature to design pattern mapping. Then, the application’s executable design patterns are created by customizing the SPL executable design patterns. This involves systematically customizing each SPL design pattern specification and executable version based on the feature to design pattern mapping and feature selection for this application. As discussed in section 4, two FSW applications were developed in detail from the FSW SPL, one of which is described in this paper. The application derivation and development process is illustrated below using the SNOE case study. The FSW SPL features are selected based on SNOE’s requirements. For example, as SNOE is only required to process a low volume of ground commands, the Low Volume Command Execution feature is selected from the Command Execution feature group in Fig. 2.

6.1 Modeling Application Level Executable Design Patterns

Next, the application level design patterns are determined from the feature to design pattern mapping. For instance, SNOE realized the Low Volume Command Execution feature. This feature is mapped to the FSW SPL Centralized Control design pattern. Therefore SNOE will realize and customize this executable design pattern. The process for deriving SNOE’s executable design patterns from the FSW SPL executable design patterns involves systematically customizing each SPL design pattern specification and executable version based on the feature to design pattern mapping and feature selection for this application. For example, SNOE uses the Centralized Control design pattern as it is mapped to the Low Volume Command Execution feature, as shown in Table 1.

First, the collaboration diagram is customized to reflect the application specific components. This involves removing optional components that are not selected, updating the component multiplicities, and selecting the appropriate variants based on the application’s features. In some cases variants are unique to the application, in which case they must be designed and developed at the application level. This process is illustrated using SNOE’s Centralized Control executable design pattern collaboration diagram shown in Fig 6. Based on SNOE’s
feature selection, it is determined that none of the optional components are utilized and consequently they are all removed. Additionally, the SNOE specific variants are selected based on SNOE’s feature selection. For example, instead of using Attitude_Control_Device_OC, the specific type of attitude control device variant is reflected with the Torque_Rod_OC. SNOE uses two torque rods, thus its multiplicity is one or many. Finally, SNOE’s unique payloads variants are included.

Next, the interaction diagrams must be customized for the application based on the feature selections for the application. As part of this process, if an SPL design pattern’s interaction diagram only contained a representative set of interactions, then the interaction diagram must be updated to reflect the precise sequence of interactions.

This process is illustrated using the Command Execution interaction diagram for SNOE. Fig. 7 depicts a subset of the SNOE specific interactions for Command Execution interaction diagram. This interaction diagram is based on the FSW SPL command execution interaction diagram in Fig. 4, which contains a representative set of object interactions. First, the SNOE specific variant components are selected based on SNOE’s pattern variability features. There are selected SPL pattern variability features for the specific type of antenna and memory storage device, see Fig. 2. The variant components are developed during the application derivation process and customized to meet the needs of the application. After the interaction diagram is updated with the application specific variants, the object interactions are also updated. In Fig. 7, the CDH Controller receives an input Event Notification to reinitialize the spacecraft’s low gain antenna. Since the specific antenna and memory storage device variants are known based on the pattern variability feature selection, the specific interactions with this output component can now be modeled in Fig. 7, which depicts different commands that can be invoked on a device. Furthermore, payload variants are unique to SNOE and thus must be modeled by the application engineer to interface to the application architecture in a predefined manner.

After this, the executable version of each design pattern must be updated for the application. This involves adding application specific states, events, and actions to the SPL level state machines based on the application’s features. For example, if the application features refines some behavior, then this can be modeled as substates. If the component must send a message to an application specific variant or if application specific logic is required then this is modeled as an action or activity within a state or transition. This process is illustrated using the state machine for the SNOE’s CDH_Centralized_Controller, which is a customized version of the FSW SPL’s CDH_Centralized_Controller (Fig. 5). SNOE’s features do not refine the overall behavior of the component, so states do not require customization. However, SNOE’s features require application specific logic to validate commands and to determine the responses to commands. The Controller must also interface to SNOE unique variants. Therefore these actions are updated for the appropriate states and transitions in the state machine.

7 Validation

The approach was validated by ensuring functional correctness of all SPL executable design patterns, as well as the patterns selected and customized for the application. The FSW SPL design patterns were validated by ensuring functional correctness of the individual executable patterns. This was done by creating test cases to cover all states, transitions, and actions for the state machines of all the components in the executable pattern. Input data to the test cases included source states and event sequences that trigger a test case. Output data included expected destination states and actions. The expected results of the test cases were compared with the behavior of the state machines. Currently, the test case creation and comparison of results is a manual process. Next, the SNOE and STEREO design patterns were individually validated. Test cases were created to cover all states, actions, and transitions for the design patterns. However, test cases are different from the FSW SPL test cases because they must test all the application customizations, including data, logic, and additional states. Then the test cases were compared with the actual behavior of the state machines. For SNOE C&DH subsystems seven design patterns were validated. The validation for STEREO’s C&DH subsystem covered a total of 10 design patterns. A subset of these patterns is shown in Table 2. It has columns.
Table 2. Subset of design patterns validated

<table>
<thead>
<tr>
<th>Design Pattern</th>
<th>FSW SPL Design Patterns</th>
<th>SNOE Design Patterns</th>
<th>STEREO Design Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchical Control</td>
<td>FSW Hierarchical Control</td>
<td></td>
<td>STEREO Hierarchical Control</td>
</tr>
<tr>
<td>Distributed Control</td>
<td>FSW Distributed Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Centralized Control</td>
<td>FSW Centralized Control</td>
<td>SNOE Centralized Control</td>
<td></td>
</tr>
<tr>
<td>Compound Commit</td>
<td>FSW Telemetry Storage and Retrieval Compound Commit</td>
<td></td>
<td>STEREO Telemetry Storage and Retrieval Compound Commit</td>
</tr>
<tr>
<td>Client Server</td>
<td>FSW Telemetry Storage and Retrieval Client Server</td>
<td>SNOE Telemetry Storage and Retrieval Client Server</td>
<td></td>
</tr>
<tr>
<td>Pipes and Filters</td>
<td>FSW Telemetry Formation Pipes and Filters</td>
<td>SNOE Telemetry Formation Pipes and Filters</td>
<td></td>
</tr>
<tr>
<td>Pipes and Filters</td>
<td>FSW Telemetry Formation Pipes and Filters with Strategy</td>
<td>SNOE Telemetry Formation Pipes and Filters with Strategy</td>
<td>STEREO Telemetry Formation Pipes and Filters with Strategy</td>
</tr>
</tbody>
</table>

for the general purpose design patterns, the SPL version of these design patterns, and the customized SNOE and STEREO versions of these patterns.

Finally, the entire SNOE and STEREO architectures, including the selected design patterns, were validated. To achieve this, a feature based validation approach based on CADeT [20] was applied. The SPL Test Model-based testing approach consists of creating activity diagrams from use cases to provide greater precision in the use case descriptions, creating decision tables to formalize the test specifications, and defining a feature based test plan that provides test coverage of all use case scenarios, all features, and selected feature combinations of a SPL. Feature-based Test Derivation consists of deriving the test specifications for the derived application, selecting the test data, and testing the application. This approach helps to reduce the overall validation effort by creating reusable SPL test assets that can be customized for SPL applications. Using this approach, 22 feature-based test specifications were created and passed for SNOE and 32 feature-based test specifications were created and passed for STEREO.

8 Discussion

This paper has described a pattern based modeling approach for SPL Engineering, which is in contrast to other SPL modeling approaches in particular the tradeoff between the development effort in domain engineering vs. application engineering.

This paper then described the application of a design pattern based SPL approach for building FSW SPL. This approach is useful in the FSW domain because architectural variability is captured at a larger degree of granularity using software architectural design patterns, thus less modeling is required during the SPL engineering phase. The tradeoff with this approach is that additional modeling is required during the application engineering phases. This tradeoff is acceptable in domains such as FSW, where modeling all possible variations during the SPL engineering phase can be time consuming and may not always be known in advance. In addition, in certain domains, application variability, such as payload variability in FSW, is mission specific and hence specific to a single application. It is thus considered an advantage to develop mission specific components during application engineering rather than domain engineering.

Using the design pattern based approach for the FSW SPL required significantly less component modeling and design during domain engineering than a component/connector based SPL approach. In the FSW SPL, during the domain engineering phase, the design pattern based approach required modeling 29 components containing representative SPL behavior, while the component/connector based SPL approach requires 53 components for all the different SPL variants. As previously discussed, the tradeoff is that additional modeling is required during application engineering. During application engineering of SNOE, 10 FSW SPL components were customized and in STEREO, 22 FSW SPL components were customized.

The Variation Point Model and SPL Architectural Design Pattern approaches are in contrast to other SPL approaches [1-4] that develop all the possible variants during domain engineering. In these cases, the application engineer would choose which variants they would like to use. The advantages of these approaches are the decrease in time to market because the application developer does not need to create a variant, only choose a variant.

The pattern-based SPL approach described in this paper addresses the problem of SPL variability by specifying this variability at a higher level of granularity through architectural design patterns. This approach requires less modeling and design during domain engineering but necessitates addi-
tional modeling and design during application engineering. The pattern-based SPL approach works well in situations where not all the variability is known during domain engineering, so that deferring decisions to application engineering is an advantage. It is also appropriate in cases where each SPL application need unique variant components, so that deferring the implementation of these components to application engineering is not a disadvantage.

9 Conclusions

This paper has described an approach for addressing SPL architectural variability at a larger degree of granularity using software architectural design patterns and compared this approach to other SPL development approaches. The key to this approach lies in modeling architectural design patterns at the domain and application engineering levels to progressively address variability within the patterns themselves and variability in the patterns selected for a member application. In addition, developing executable versions of these patterns allows the validation of the patterns and the architectures they are incorporated into.

The approach described in this paper has several benefits. First, the approach provides domain specific patterns that can be applied to FSW product lines. Second, executable versions of the patterns are developed to allow the patterns to be validated. Third, providing domain specific design patterns provides a systematic approach, which leverages best design practices, for incorporating these patterns into the SPL architecture and its member applications. Furthermore, executable architectures produced from these patterns using executable state machines can be validated during the design phase for functional correctness.

For future work, this research can be applied to other SPLs and to other application domains to demonstrate applicability across domains. The validation process described in this paper is manual. Future research will seek to automate the process to improve the practicality of the approach.

10. References

[9] E. Gamma, R. Helm, R. Johnson, and V. John, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.