Mainstream forces are driving Systems and Software Product Line (SPL) Engineering approaches to take a more holistic perspective that is deeply integrated into the systems and software engineering lifecycle. These forces illustrate that SPL challenges will not be solved at any one stage in the product engineering lifecycle, nor will they be solved in independent and disparate silos in each of the different stages of the lifecycle. This paper describes BigLever Software’s response to these forces – an SPL Lifecycle Framework. The motivation for this technology framework is to ease the integration of tools, assets and processes across the full systems and software development lifecycle. The goal is to provide product line engineers with a common set of SPL concepts and constructs for all of their tools and assets, at every stage of the lifecycle, and to assure that product line development traceability and processes flow cleanly from one stage of the lifecycle to another.

1. Introduction

According to Geoffrey Moore’s classic hi-tech marketing book *Crossing the Chasm*, widespread adoption of a new hi-tech offering requires more than just a great concept, technique, technology or product. The mainstream majority of users require a complete solution. Although a small percentage of potential users – the Innovators and Early Adopters according to Moore – are willing to take on the task of extending new ideas and technology into a complete and workable solution, the majority of users that make up the mainstream – the Early Majority and Late Majority – are simply unwilling or unable to dedicate the resources needed to innovate complete solutions out of promising concepts, techniques, technology and tools[1].

These mainstream forces from the early majority and late majority contingents now require SPL approaches to provide holistic solutions that are deeply integrated into the full systems and software engineering lifecycle. We describe here BigLever Software’s response to these forces – the SPL Lifecycle Framework for the full systems and software product line engineering lifecycle.

The SPL lifecycle framework is a key ingredient to a complete, out-of-the-box SPL solution that enables easy adoption by mainstream organizations. It provides an integration framework for software engineering tools, assets and processes across the full systems and software development lifecycle. The SPL framework offers engineers and managers a common set of SPL concepts, constructs and capabilities for all of their existing tools and assets, at every stage of the lifecycle.

When used in conjunction with BigLever’s 3-Tiered SPL Methodology[2], the SPL framework offers a straightforward transition path from legacy approaches to a complete SPL solution. The SPL lifecycle framework has enabled mainstream organizations with some of the largest, most sophisticated and complex, safety-critical systems ever built to adopt the SPL approach.
2. Importance of the SPL Engineering Lifecycle

A fundamental tenet of software product line engineering is that product line variation must be simultaneously managed along two dimensions, time and space[3]:

- variation in time, to manage asset evolution (i.e., multi-baseline management)
- variation in space, to manage diversity within assets and products in the product line (i.e., multi-product management)

Practical experience with numerous commercial SPL deployments at BigLever Software has illuminated that there is an equally important third dimension that must be addressed in order to satisfactorily provide a complete SPL solution for mainstream product line engineering organizations:

- variation in the lifecycle, to provide consistency and to manage traceability among asset variations in different lifecycle stages (i.e., multi-phase management)

These three dimensions are illustrated in Figure 1, multi-product in the domain space, multi-phase across the lifecycle, and multi-baseline in time.

![Figure 1. The Three Dimensions of a Complete SPL Solution](image)

The importance of the additional multi-phase lifecycle dimension for providing a complete end-to-end solution to mainstream organizations can be attributed to:

- Business process management. Well defined business processes are common in mainstream organizations, often due to large size or external contractual or legal governance. As a result, the engineering lifecycle is usually explicitly defined by these organizations, within each lifecycle stage and across the boundaries between each of the stages.
- Systems and software engineering. Software is often engineered within the larger context of a system. Embedded software is the obvious example, though other common examples include enterprise and web-based software that is designed in the specific context of server, network or specialty hardware and IT system architectures. As a result, the engineering lifecycle includes the combined systems and software perspective for requirements, architectures, models, documentation, test plans and so forth, rather than just a narrow software-only product line perspective.
When transitioning to an SPL approach, a complete solution must retain the holistic lifecycle focus and capabilities. That is, a complete solution must simultaneously manage systems and software product line commonality and variability in all three dimensions of time, space and lifecycle.

2.1. The Need for an SPL Lifecycle

One of the key capabilities of a good software engineering lifecycle solution is traceability[4]. For example, every requirement should be traceable to one or more design elements that satisfy that requirements and each design element should be traceable back to one or more requirements that it satisfies. Each design element should be traceable forward to its implementation and vice versa. Each requirement should be traceable to one or more test cases that validate whether or not the requirement is satisfied in the final product.

For the lifecycle of an individual product, traceability is represented as a static set of relationships among the assets in the lifecycle (e.g., requirements, architecture, design, source code, test cases, documentation). Of course, as the software assets evolve over time, the traceability relationships also evolve. In fact, traceability relationships are intended to facilitate evolution by indicating which related assets need to change whenever any asset is modified. Thus, traceability for a single product applies to the multi-baseline and multi-phase dimensions of Figure 1.

When traceability is applied to a product line rather than an individual product, the third dimension – multi-product – must now be accommodated as well. Similar to the case of an individual product, traceability among the parts of assets that are common to all products is represented as a static set of relationships. However, for the parts of the assets that vary from product to product, the traceability relationships in and out of these “variation points” must also vary. This is where most SPL concepts, technology tools and techniques fall short.

A review of the proceedings from the first eleven Software Product Line Conferences illustrates a wide range of SPL concepts, techniques, technology and tools[5]. Each of these are almost exclusively applied to a single stage in the development lifecycle, without consideration of how they might be applied in combination with other concepts, techniques, technology and tools in other lifecycle stages. For mainstream organizations looking for a complete SPL solution, this situation poses a prohibitive adoption barrier.

For example, imagine a requirements engineering team has embraced an SPL requirements management technique based on tagging requirements in a requirements database with attributes that differentiate feature variations in requirements. The design team has adopted a UML tool and has embraced inheritance as the mechanism for managing SPL design variations. The development team is using a FODA feature model drawn in a graphical editor, plus #ifdefs, build flags and configuration management (CM) branches to manage SPL implementation variations. The test team had adopted clone-and-own of test plan sections, stored in appropriately named file system directories to manage their SPL test plan variations.

Now imagine what would be needed to create a complete SPL lifecycle solution that integrates into a larger business process model. How do the requirements database attributes and tagged requirements relate and trace to the subtypes and supertypes in the design models? How do these attributes and supertypes relate and trace to the #ifdef flags, CM branches, FODA features, and test case clone directories?

Without answers to these questions, it is not possible to define a lifecycle process that flows cleanly from one stage of the lifecycle to another. Trying to translate between the different representations and characterizations of features and variations creates dissonance at the boundaries between stages in the lifecycle.

This example illustrates that a complete SPL solution cannot be solved with concepts, technology tools and techniques at any one stage of the lifecycle. Nor can it be solved with a combination of disparate point solutions at different stages.
2.2. Need for a Unified Systems and Software Product Line Engineering Lifecycle

In the case where software is engineered in the context of a larger system, the early stages of the lifecycle include assets that focus on both systems and software, such as high-level system requirements. For SPL, it is essential to capture and express the product line variations from a systems and software perspective, so that traceability within the lifecycle is managed from the earliest stages. In mainstream engineering organizations, we often find that systems requirements are perceived as one of the most critical parts of a complete SPL solution and that this is the stage of the lifecycle that must be addressed first in a transition.

3. The SPL Lifecycle Framework

In response to the needs of mainstream engineering organizations for a complete systems and software product line engineering solution, BigLever Software created the SPL Lifecycle Framework. This framework supports all three of the SPL solution dimensions – multi-baseline in time, multi-product in the domain, and multi-phase in the lifecycle – as illustrated in Figure 1. The motivation for this technology framework is to ease the integration of development tools, assets and processes across the full systems and software engineering lifecycle.

The goal is to provide product line engineers with a common set of SPL concepts and constructs for all of their tools and assets, at every stage of the lifecycle. Furthermore, the goal is to apply these concepts and constructs uniformly, so that traceability and development processes flow cleanly from one systems and software lifecycle stage to another – for systems analysts, requirements engineers, architects, modelers, developers, build engineers, document writers, configuration managers, test engineers, project managers, product marketers and so forth.

The SPL lifecycle framework is an integration framework for software development tools and assets. It is narrowly focused on the SPL issues of variation management in the product space and variation management in the lifecycle phases (and enabling the third dimension of time to be managed by conventional configuration and baseline management).

The SPL framework provides product line engineering organizations the following[6,7]:

- A single feature model to uniformly express product diversity, for all assets in all stages of the systems and software development lifecycle including requirements, architecture, models, design, source code, test cases and documentation.
- A single variation point mechanism that can be uniformly applied to all tools and their associated assets in all stages of the systems and software development lifecycle, including tools for requirements management, model-driven development, source code development, test case development, configuration management, build automation, change management and document development.
- A single, automated product configurator that uniformly assembles and configures assets from each stage of the development lifecycle to automatically produce all the products in a product line.

An example SPL engineering environment using the SPL lifecycle framework is illustrated in Figure 2.
The core of the framework is shown as the shaded box in the center of the diagram. It contains:

- feature model and feature profiles (profiles are instantiations of the feature model with feature options selected)
- product configurator that automatically configures all the assets for a product instance based on the feature selections in a feature profile
- production line development environment for developing feature models and feature profiles, running the product configurator to instantiate products, developing and managing variation points in the SPL reusable assets, and other SPL-specific engineering tasks.

Figure 2. The Software Product Line Lifecycle Framework

Shown to the left of the framework is an example set of development tools that have been “plugged in” to the framework. Each organization using the SPL framework will have its own preferred set of development tools, so the framework has a tool integration interface and many off-the-shelf integration “bridges” available.

On the far left of the diagram is the collection of SPL core assets for the full SPL engineering lifecycle – requirements, design models, source code, user documentation and test cases. The SPL variations within the reusable core assets are all implemented using common variation point concepts and constructs provided by the SPL framework. The variation points can automatically configured by the framework’s product configurator, based on the feature selections in a feature profile.

On the far right of the diagram is the full set of product instances that can be produced by the product configurator in the framework. There is a one-to-one correspondence to the feature profiles and the product instances – each feature profile is used by the product configurator to automatically configure the corresponding product instance. Note that all the assets from the full development lifecycle are produced for each of the products in the product line.
The underlying SPL technology for the framework, such as the configurator, feature profiles and variation points, is described in [6,7]. Similarly, the underlying SPL methodology is described in [2,7].

The SPL framework provides the critical lifecycle dimension needed for a complete SPL solution: (1) the common set of SPL concepts and constructs for tools and assets at every stage of the lifecycle, so that (2) traceability and processes flow cleanly from one lifecycle stage to another.

4. Common SPL Constructs and Concepts Across the Lifecycle

To overcome the problem of dissonant SPL concepts, technology, tools and techniques imposing disparate SPL silos in the development lifecycle, the SPL framework provides a complete SPL solution by introducing common constructs and concepts across the full systems and software lifecycle. The common SPL concepts and constructs are provided by way of tool integration and asset integration into the SPL framework.

4.1. Tool Integration

The purpose of tool integrations into the SPL framework is simply to make conventional development tools “SPL-aware”. This means making a tool aware and capable of working on reusable and configurable SPL core assets that contain internal SPL variations rather than merely working on conventional assets for one-of-a-kind applications.

The central concept in making any software development tool SPL-aware is the variation point. Variation points are the encapsulated locales within an asset that can be instantiated in different ways by the product configurator, based on feature selections in the feature profiles[6]. For a development tool to be SPL-aware, it must be able to:

- identify and display variation points
- create and modify variation points (see next subsection on Asset Integration)
- aid the framework’s product configurator in instantiating variation points during product configuration
- display the instantiated representation of a variation point, as well as all the uninstan-
tiated variability
- aid the framework’s production line development environment with semantic checks on variation points, impact analysis, statistics, queries and other SPL management operations

The SPL framework provides a tool integration API for making existing software development tools SPL-aware.

Specific characteristics of a tool and its associated assets determine the type of integration that is required. When a tool’s assets have a text-based representation in the file system, the framework can often provide all the variation point capabilities, without the need for an explicit tool integration.

When a tool’s assets are maintained in a proprietary internal representation, a deeper two-way integration is often required, where the tool is made SPL-aware and the framework is made tool-aware. For example, in Figure 2, the IBM Rational DOORS® requirements management tool uses a database for its proprietary internal representation of requirements (see first listing under Example Tools). A two-way bridge was created as a dual plugin between DOORS and the SPL framework, to make them mutually aware, so that operations and data can be exchanged to collaboratively perform SPL engineering operations.
The integration bridges are independently installable options in the framework. When an organization transitions to an SPL approach using the SPL framework, they identify preexisting bridges to match their existing toolset and install them into the framework, as in the example in Figure 2. Multiple commercial tool vendors have recognized the importance of the SPL lifecycle framework and are working with BigLever to provide bridge integrations for their tools across the full lifecycle.

4.2. Asset Integration

The purpose of asset integration into the SPL framework is to enable conventional one-of-a-kind software assets to become reusable and configurable SPL core assets. The key to asset integration is adding the variation point concept and constructs into the asset structure[6]:

- identify which asset constructs can become variation points
- define variation point encapsulations to contain the optional and alternative variants for the variation point, as well as the variation point logic to drive the variation point instantiation
- define the representation for variation point instantiations

To achieve the objective of common SPL concepts and constructs across all stages of the lifecycle, the variation point implementation in all assets must adhere to the variation point meta-model for the framework, as briefly outlined in the three bullets above.

For example, with file and text based source code assets, the constructs that can become variation points are typically directories, files, text blocks, text patterns and text tokens. Directories make good variation point encapsulations to hold file variants and variation point logic files. The variation point instantiations can simply be files and directories.

In another asset integration example, variation points in a UML™ model could be any of the model elements. In the IBM Rational Rhapsody® integration, UML “tags” within a model element are used to hold the variation point logic and the optional and alternative variants for the model element. The variation point instantiation for a model element is reflected in the executable code that Rhapsody generates off of a model element variation point[8].

In the DOORS requirements integration into the framework, a requirement and its descendants can become a variation point. The instantiation logic for a variation point is contained in a DOORS attribute. For requirements alternatives in a variation point, child requirements are specially tagged as variants for the parent requirement. The variation point instantiation is displayed in a separate database column.

4.3. Shared SPL Concepts and Constructs from the Framework

The SPL framework provides a single feature model and a single set of feature profiles that are shared across all the integrated tools and assets in a product line. The single product configurator in the framework provides a common instantiation semantics and mechanism, for all the integrated tools and assets. The framework’s SPL development environment provides a single, common administrative and development interface for managing SPL-specific engineering tasks for all the tools and assets across the full development lifecycle. These common concepts and constructs are provided by BigLever Software Gears™.
5. **Complete and Consistent End-to-End SPL Lifecycle**

The uniformity of concepts and constructs provided by the SPL framework allows traceability and processes to flow cleanly among the different lifecycle stages, without the dissonance created by independent technology in different lifecycle stages.

### 5.1. Traceability

There are three simple heuristics for defining and managing traceability among the assets in different stages of the lifecycle.

- **Traceability among the common parts of the assets** is managed identical to the way it is managed with one-of-a-kind products. Common requirements trace to common design elements, which trace to common source code elements, and so forth.

- **Traceability among the variation points in the assets** is managed similarly. Variation points in requirements trace to variation points in design elements, which in turn trace to variation points in source code, and so forth.

- **If there is a traceability relationship between a common artifact in one stage and a variation point in another stage (or vice versa),** it is most likely an error in the traceability relationship. This type of traceability consistency analysis can be automated by tools in the framework.

Traceability with the SPL framework is remarkably simple compared to the intractable complexity described earlier when trying to define traceability relationships among disparate SPL point solutions in different lifecycle silos.

### 5.2. Process Flow

The flow of software development processes among different lifecycle stages closely mirrors traceability. For example, passing a new set of requirements to an architecture and design team leads to new design models, with traceability relationships between the requirements and design. If the intended traceability in the requirements is difficult to extract and express, then there will be dissonance in the process between the requirements and design stages in the lifecycle.

Thus, the shared SPL concepts and constructs in the different lifecycle stages, plus the simplicity of traceability shown in the previous section translates to simplicity in the process at the lifecycle boundaries.

Furthermore, since the framework feature model and feature profiles are shared across all lifecycle stages, assets in all stages are constructed under the same SPL feature set. This makes it easier to understand assets and their variations in multiple stages when processes depend on it.

---

6. **Conclusions**

With the BigLever SPL Lifecycle Framework providing a common set of SPL concepts and constructs at every stage in the systems and software engineering lifecycle, a software product line can be defined, viewed and managed as a *single production system*, rather than a collection of disparate silos for each of the lifecycle stages. This single production system represents a complete SPL solution that can be readily adopted by mainstream development organizations.
References


