27th Annual INCOSE International Symposium (IS 2017)
Adelaide, Australia, July 15-20, 2017

An Enterprise Feature Ontology for Feature-Based Product Line Engineering

Dr. Charles Krueger
BigLever Software
ckrueger@biglever.com

Dr. Paul Clements
BigLever Software
clements@biglever.com

Abstract. Feature-Based Software and Systems Product Line Engineering ("Feature-Based PLE") has emerged as a modern, repeatable, codified, and proven-in-practice specialization of generic PLE practice. Feature-Based PLE involves automation-supported configuration of engineering and operations artifacts from across an enterprise, to reflect the feature choices embodied by a product. Configuration of artifacts based on feature choices is a powerful narrative, but what actually constitutes a feature? A feature is a distinguishing characteristic that sets products apart (Kang et al. 1990) but in practice this concept encompasses a broad spectrum of granularity. Stakeholders who are concerned with the finest-grained differences are very different from those who care about the coarsest-grained differences. This paper presents an ontology for features that is capable of supporting an entire enterprise of product line engineering and business operations, and shows how various stakeholders throughout the enterprise can interact with these different kinds of features.

Introduction

Product line engineering (PLE) is an approach for engineering a portfolio of related products in an efficient manner, taking full advantage of the products' similarities while respecting and managing their differences. By "engineer," we mean all of the activities involved in planning, producing, delivering, deploying, sustaining, and retiring products.

Born in the 1980s in the software field, but now having grown well beyond those early roots, PLE offers large savings observed from engineering the whole family rather than separately engineering each member. Numerous case studies (Clements and Northrop 2002, van der Linden et al. 2007, Software Engineering Institute "Catalog" 2017, SPLC 2017) show that exploiting the commonality throughout the products' total life cycles can return substantial improvements in time to market, cost, portfolio scalability, engineer productivity, and product quality (Software Engineering Institute "Benefits and Costs"); no other engineering paradigm shift has, to our knowledge, brought about results that rival these.

Recently, a form of PLE known as *Feature-Based Software and Systems Product Line Engineering* ("Feature-Based PLE") has emerged as a modern, repeatable, codified, and proven-in-practice specialization of generic PLE practice. Supported by industrial-strength automation and methodology, Feature-Based PLE is the subject of an upcoming ISO standard[1] that is in progress with involvement and support from INCOSE through its Product Line Engineering International Working Group (INCOSE 2016). Feature-Based PLE involves automation-supported configuration of engineering artifacts to reflect the feature choices embodied by a product. Configuration of artifacts based on feature choices is a powerful paradigm, and there are many successful

[1] ISO/IEC 26580, "Methods and Tools for the Feature-Based Approach to Software and Systems Product Line Engineering," in progress.

applications of Feature-Based PLE in the literature[2] (Clements et al. 2013, Dillon et al. 2012, Flores et al. 2012, Gregg et al. 2015).

As implied by the name, the notion of *feature* lies at the conceptual heart of Feature-Based PLE. Briefly, a feature is a distinguishing characteristic that sets products in a product line apart from each other (Kang et al. 1990). But what actually constitutes a "distinguishing characteristic?" In a thought experiment, imagine performing a "diff" operation across all of the engineering artifacts that represent all of the product instances in a product line. It would return requirements objects, model elements from design specs, test cases, individual lines of software, mechanical parts in a Bill of Materials, sections or paragraphs or even individual words in a user's manual, slides in training courseware, and much more – all of the places where the digital engineering representations of any two products differ from each other.

In a product line comprising thousands to tens of thousands of instances such as in the automotive domain, for example, there may easily be millions of these differences. While they are undeniably "distinguishing characteristics," albeit tiny ones, and undeniably important to manage correctly, it becomes immediately clear that they cannot be features: No organization could create and manage a bank of millions of features[3]. More precisely, there cannot be a one-to-one correspondence between a feature and a variation in an engineering artifact. We need a more abstract concept to power the Feature-Based PLE approach in a practical and powerful way.

The PLE literature is rife with examples of feature models, typically shown in a notional box-and-line representation. For simple product lines, simple feature models such as these can be sufficient for managing the feature-based variation in the engineering assets across the lifecycle. It will turn out that this simplistic notion of feature is not wrong (this ontology will have a place for box-and-line diagrams as well) but in practice, especially in the realm of very large product lines built by very large organizations, it just scratches the surface of what is needed.

Product line enterprises are now recognizing that feature-based variant management also has value in organizational operations "on the wings" of the engineering lifecycle, in areas such as product marketing, portfolio planning, manufacturing, supply chain, product sales, product service and maintenance, resource planning, financial projections, and much more. The implication is that thousands of non-engineering users within an enterprise need different views and scenarios to interact with an all-embracing feature representation.

This paper presents an enterprise *ontology* for features that is suitable for managing feature-based product line engineering and operations in the largest and most complex product line organizations. An ontology is an "explicit formal specification of the terms in a domain and relations among them" (Gruber 1993), in order to "share common understanding of the structure of information among people or software agents" (Noy 2016). The people are those who will use the Feature-Based PLE paradigm. The "software agents" to inform include the PLE tools that are the technological engines of the Feature-Based PLE paradigm.

This enterprise feature ontology has eight layers of abstraction. Each incrementally constrains the complexity, establishes the level of discourse, and targets a specific set of roles in the business enterprise that will use it. The ontology is independent of any particular PLE-enabling tool, but to

[2] Feature-Based PLE is synonymous with "Second Generation" PLE (2GPLE), the term used by some of the references.

[3] Ten million variations means ten million places where an in-or-out decision must be made, which defines a space of $2^{10,000,000}$ combinations. By comparison, 2^{33} is the number of humans living on Planet Earth, and 2^{280} is the number of atoms in the visible universe (Villaneuva 2009).

illustrate how the concepts can be made concrete in PLE tooling we have illustrated it using the Gears PLE technology (BigLever Software 2017).

Feature-Based Product Line Engineering

PLE as a Factory

Feature-based PLE uses a reference model that is based on a factory metaphor. The PLE factory produces digital assets across the systems engineering and software engineering lifecycle for products in a product family, analogous to a conventional factory producing physical assets and products. All development happens inside the factory; the output of the factory is for validation and delivery.

The Feature-Based PLE Factory shown in Figure 1 has three distinct layers.

- The base *Technology* layer comprises the tools and technology of the factory infrastructure. Think of this as the fully functional factory without any of the people inside to run the factory.
- The middle *Technical Organization Management* layer focuses on the people, roles, and processes that operate the PLE factory. In combination, the base Technology layer and the middle Technical Organization Management layer provides a fully operational Feature-Based PLE Factory capable of producing the products in a product line portfolio.
- The top *Business Organization Management* layer focuses on the people, roles, and processes that utilize and leverage the PLE Factory to achieve the business objectives of the enterprise. Using the analogy to a conventional factory, the top Business Organization Management layer provides guidance and support for the executive leadership working the office high-rise that overlooks the factory.

Because features reside in the Technology layer, the remainder of the paper will focus on that.

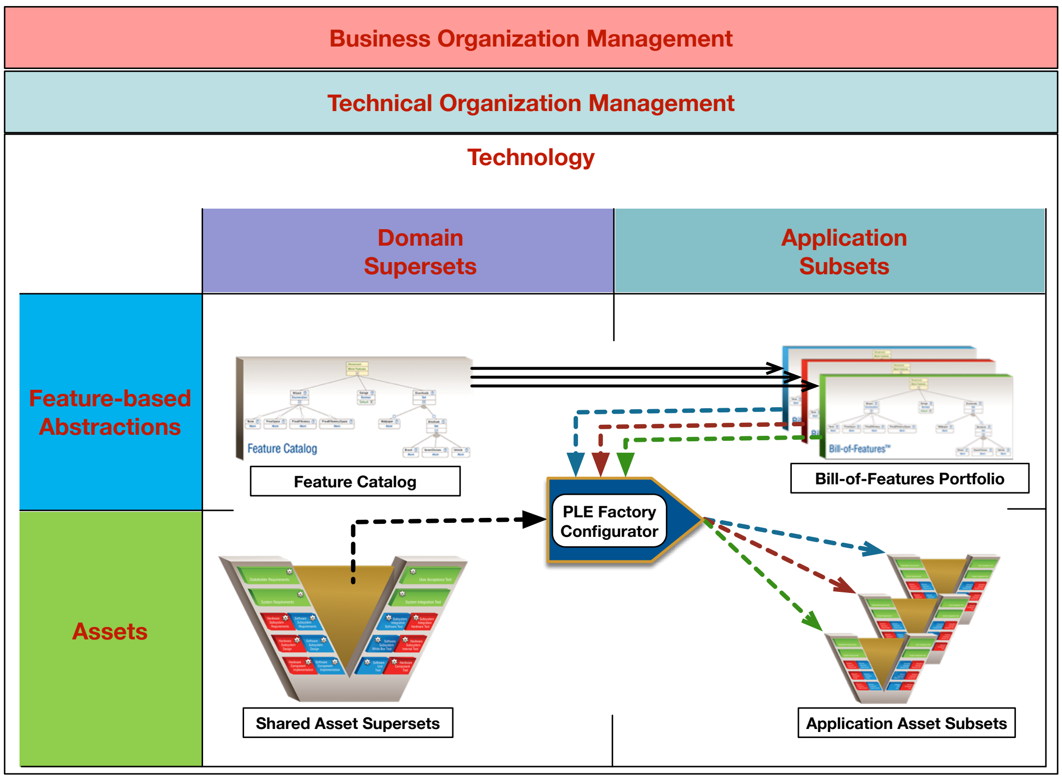


Figure 1: Reference model for the PLE Factory in Feature-Based PLE

Technology Layer Content

The components of the PLE Factory are as follows:

- A product line's Feature Catalog is a model of the collection of all of the feature options and variants that are available across the entire product line. Feature Owners, Feature Architects create their respective sections in the feature catalog.
- A Bill-of-Features is a specification for a product in the product line portfolio, rendered in terms of the specific features from the Feature Catalog that are included in the product. In other words, it is a feature-based product specification, defined as an instantiation of the available feature choices in the Feature Catalog. The Bill-of-Features Portfolio is the collection of Bills-of-Features for the entire product line. Portfolio teams, typically working with product marketing teams, create Bill-of-Features for product families and "flavors", based on the features available in the Feature Catalog.
- Shared assets are the digital artifacts associated with the systems and software engineering lifecycle of the product line. Shared assets can be whatever digital artifacts compose a part of a delivered product or support the engineering process to create and maintain a product. They typically include requirements, source code, test cases, user documentation, a bill of materials (parts lists) and more. Shared assets can include, but are not limited to, requirements, design specifications, design models, source code, build files, test plans and test cases, user documentation, repair manuals and installation guides, project budgets, schedules, and work plans, product calibration and configuration files, data models, parts lists, and more. Assets in PLE are engineered to be shared across the product line. In Feature-Based PLE, shared assets are maintained as *supersets*; that is, any content needed by any of the products can be found in the superset. The supersets include *variation points*, which are declarations that specify under what feature choice combinations a specific piece of content is needed and in what form it is needed. Variation point content will be included in a product's digital assets if that feature or feature combination has been chosen, and omitted otherwise. Asset engineers create variation points in their subsystem assets, based on the Features available in the Feature Catalog. Asset engineers typically specialize into their own specific disciplines: Requirements Engineers, System Modeling Engineers, Test Engineers, BoM Engineers, Document Engineers, and so forth.
- The PLE Factory Configurator is the mechanism that automatically produces application assets for the digital twin of a specific product. In Feature-Based PLE, the configurator is an automated tool, as opposed to a manual process. It performs its task by processing the Bill-of-Features for that product, and exercising the shared assets' variation points in light of the feature choices in that Bill-of-Features. The configurator provides the abstraction-driven automation the eliminates the labor intensive and error-prone activity of manually assembling and modifying engineering assets for the digital twin for each product in the product line.
- Application Asset Subsets are product-specific instances of the Shared Asset Supersets, automatically produced by the PLE Factory Configurator. Application Asset Subsets are validated and delivered to the next lifecycle phase (e.g., manufacturing), to the customer, or to the market. Suppliers, testers, manufacturing, sales and more use the generated assets, configured by the Bill-of-Features for a system or product family

In the factory, Shared Asset Supersets are configured by the PLE Factory Configurator based on a Bill-of-Features in the Bill-of-Features Portfolio, derived from the Feature Catalog, to produce an Application Asset Subset. The complete collection of digital engineering assets across the systems engineering and software engineering and operations lifecycle for a single physical product is sometimes referred to as the "digital twin" of that product. Thus the Feature-Based PLE Factory in this reference model gives birth to the digital twin for each product in a product line portfolio.

Features Are Abstractions

In the Introduction we asserted that features cannot correspond one-for-one with variations in product artifacts, which we can now call Application Asset Subsets. In order for a Feature Catalog to be tractable to represent and manage, a feature needs to be more powerful than a switch that turns bits of content on and off in each Shared Asset Superset. Therefore, in order to realize the Feature-Based PLE narrative (feature choices are used to configure engineering artifacts), there must be a one-to-*many* correspondence between features and engineering artifact variation.

Happily, another word for a one-to-many mapping is *abstraction* (Saitta et al. 2013), which is exactly the concept we need: Features are abstractions of variations; one feature can "drive" multiple artifact-level variations. This one-to-many mapping matches our engineering intuition about features: If a capability is included in a system, then requirements, design elements, source code, test cases, user documentation and more, *all* corresponding to that capability, should be included in the product's digital twin. If the capability is omitted, then all of that material should be omitted.

It also matches our practical intuition about features. To elicit the distinguishing characteristics among products, we would not expect our product portfolio experts to tell us about differences in lines of code or test cases. Rather, we would expect to hear about more abstract differences expressed in terms of capability, function, usage environment, etc.

Primitive Features

We have now arrived at a notion of a feature that matches common usage in the PLE literature. A feature is an abstraction that can describe variations among products in way that applies across all engineering disciplines and their associated artifacts. In this way, features can serve as the lingua franca for engineers, giving them a common terminology with which to describe, manage, and implement differences among the products in a product line (Krueger and Clements 2013).

In practice, features are not captured as separate items in a list, but rather in a structure, typically a tree structure such as the one in Figure 2 (truncated on the right due to page size). The tree is a decision tree that lays out choices to be made to describe a system. This *feature model* captures some kinds of relationships among features. In Figure 2, features have types – Boolean, Enumeration, Set., etc. – that capture these relationships. For example, the feature LockoutDeterrent is a Boolean, meaning it can be present or not, with no further elaboration. The feature LockStatusIndication is an Enumeration, meaning that exactly one of its children, no more and no less, is available to a product. The feature RequestSource is a Set, meaning that any number of its children are available to a product. And so forth.

Making features abstract with respect to specific variation points in specific engineering artifacts reduces the number possibilities by orders of magnitude. A product line as large as the portfolio of a company like General Motors may comprise on the order of ten thousand features like the ones we are describing here (Wozniak and Clements 2015). For a company as large as General Motors, with thousands of engineers working in the product line context, ten thousand is a manageable amount.

It is manageable, that is, as long as they are not all contained in a single monolithic model. We need a way to identify and work with modular subsets of features.

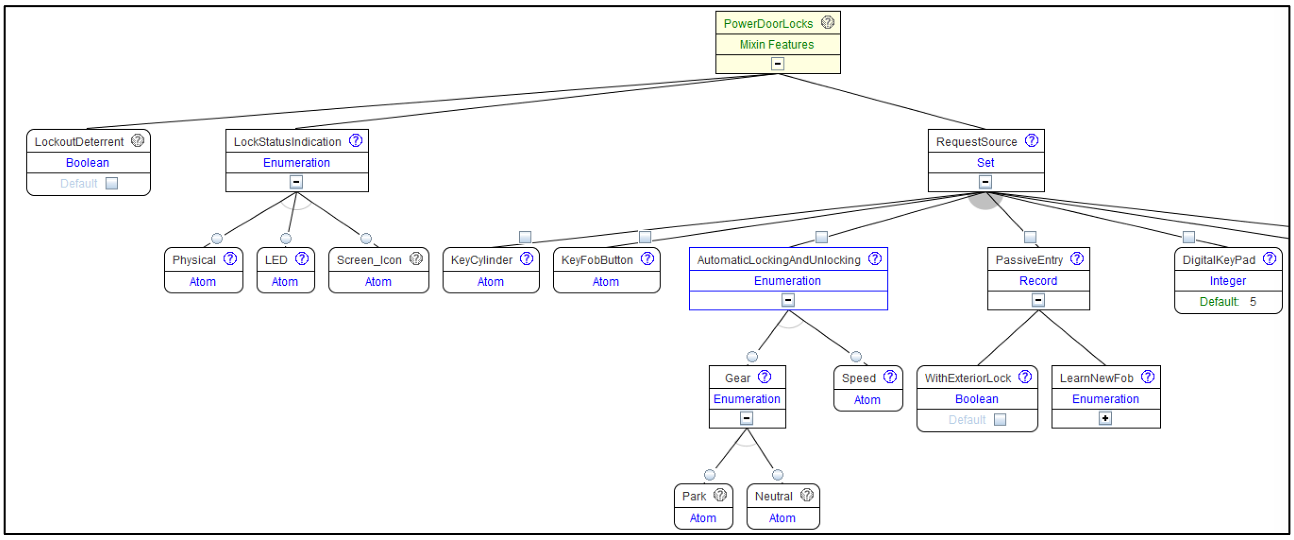


Figure 2: Feature model (partial) for an automotive Power Door Locks system

Modularizing Feature Models

Just as large software systems are divided into modules, to facilitate ease of change and productive work by separate members of a large team (Parnas 1972), so it is with feature models. We need a way to decompose a highly undesirable monolithic feature model into cleanly packaged subsets, to achieve exactly the same purposes: To achieve ease of change and to enable productive work by many members of a large team.

In the Gears environment, these modular packages of features are called *mixins*. The name comes from a computer language concept for aggregating multiple data definitions together. A PLE factory will include the aggregation of many mixins.

Mixins have names and, in practice, generally correspond to specific capabilities, or subsystems, or some other generally acknowledged unit of system decomposition. Mixins may also be used to distinguish between customer-facing features (for example, would you like a cruise control on your car that detects the car ahead of you and keeps you from running into it?) and inward-facing or implementation-oriented features (to detect a car in front of us, shall we use a camera on the front bumper, a LIDAR sensor, or a short-range radar?). Both decisions represent important and legitimate feature choices, but are meaningful to different audiences, and so it is convenient to put them in different feature models. Assuming any of the inward-facing choices would do the job, the actual decision there is based not on functionality, but on the achievement of quality attributes: *this* sensor costs less, but *that* sensor weighs less and produces less heat; however we happen to have a warehouse full of this *other* sensor.

Mixins partition the feature catalog space into multiple modular feature models. Just as dividing software into parts requires keen architectural insight and ability, defining the scope of a mixin requires the same kind of architectural thinking. Just as a software module should have high cohesion, features in a mixin should be similarly cohesive, and a mixin should be assigned to an appropriate subject matter expert (or team) to own and manage its content and evolution.

Typically, a particular shared asset, such as a requirements module, source code component, or test suite, won't need all of the features to configure its variation points. Instead, one or a few mixins will often suffice. Later in the ontology, we'll see how to make use of this.

Feature Profiles

A feature model such as one in Figure 2 only lays out choices that are available. Someone has to actually choose, and we need a way to record the choices.

We could let people browse through the feature model and choose any combination of features present. However, this quickly reveals itself to be an undesirable choice. The quite modest feature model in Figure 2 lends itself, it turns out, to *over twenty-one thousand distinct sets of choices!* That is, if you showed Figure 2 to twenty-one thousand people and asked them to make selections, it's possible that every one of them would produce a set of choices different in some way from everyone else's.

We would never want to entertain the possibility of 21,000 implementations of this small feature model. We would never ask our engineers to design and build that many; we would never ask our QA group to prepare to test that many. Thus, we don't want to let people pick and choose among individual features. Rather, we want to offer only those sets of pre-packaged choices that (a) contribute to products that are technically and economically feasible to engineer and build; and (b) contribute to products that a customer or the market would actually want to buy. We call these pre-packaged sets of choices *feature profiles*.

By contrast, a feature profile represents a unique and *supported* configuration of a feature model. These, then, are the combinations we are willing to build and test and offer to the product line at large – not just any combination at random, but a set that has been identified with analysis and forethought.

In Gears, Feature Profiles are encapsulated into the same mixin as the feature model that they configure. Thus, the feature profiles represent the valid and limited configurations offered for that mixin – that is, for that subject matter area.

Figure 3 illustrates a feature profile for the feature model shown in Figure 2. Graphically, it replicates in Figure 2 except that now buttons and check boxes are filled in, indicating the choices made. For example, "LED" is the choice for LockStatusIndication, and RequestSource includes "KeyCylinder," "KeyFobButton," "AutomaticLockingAndUnlocking," and "PassiveEntry," but not "DigitalKeyPad." "AutomaticLockingAndUnlocking" will be by "Gear," rather than "Speed," and the gear of interest is "Park," not "Neutral."

Mixin as a Higher-Order Feature Abstraction

The feature model and the list of named feature profiles that comprise a mixin represent a higher order feature abstraction in our feature ontology. A named mixin (Power Door Locks in our example) becomes a single higher-order feature that encapsulates and hides the level details of its primitive feature model tree. Each named feature profile in the mixin becomes a single higher-order choice and the list of named feature profiles become a higher-order enumeration of mutually exclusive choices for the higher-order mixin feature. For example, our Power Door Lock higher-order mixin feature might have three mutually exclusive profile choices, Manual, Automatic, and Passive. In our experience it is not unusual to see a mixin with up to 100 or so features offered with a dozen or less feature profiles, thus reducing the available choices by orders of magnitude.

The roles responsible for engineering the feature mixins are typically subject matter experts for a specific area of capability within a product. They may collaborate with the portfolio planning organization to determine which differentiating characteristics should be offered as primitive feature choices. The list of feature profiles that should be offered of also determined and defined by the subject matter experts collaborating with the business to determine which offerings provide the best economic return for the enterprise.

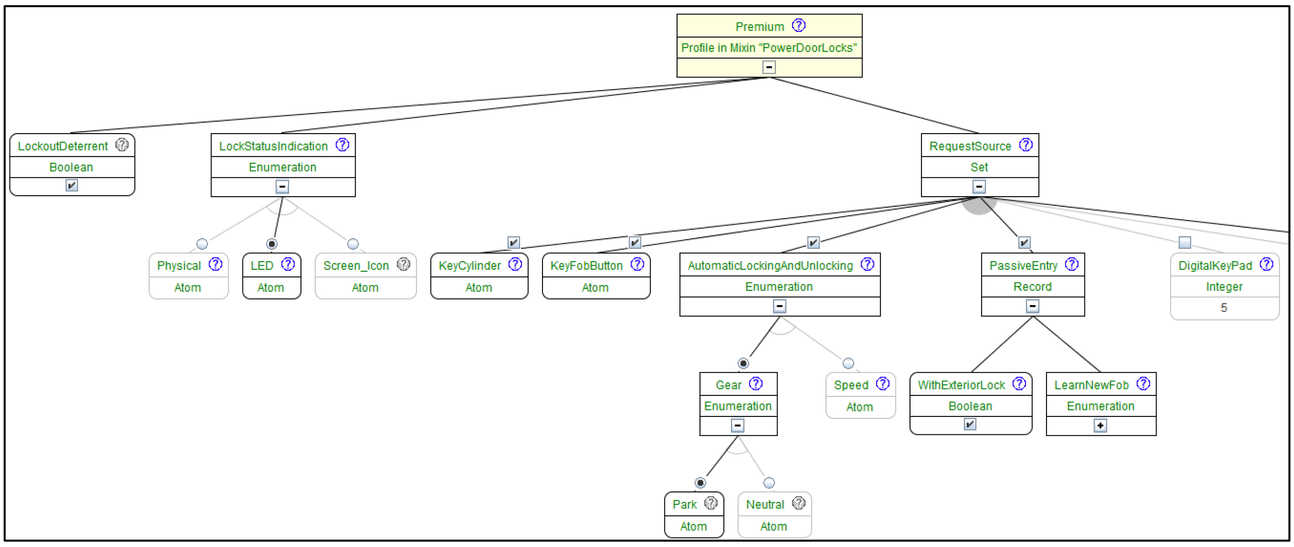
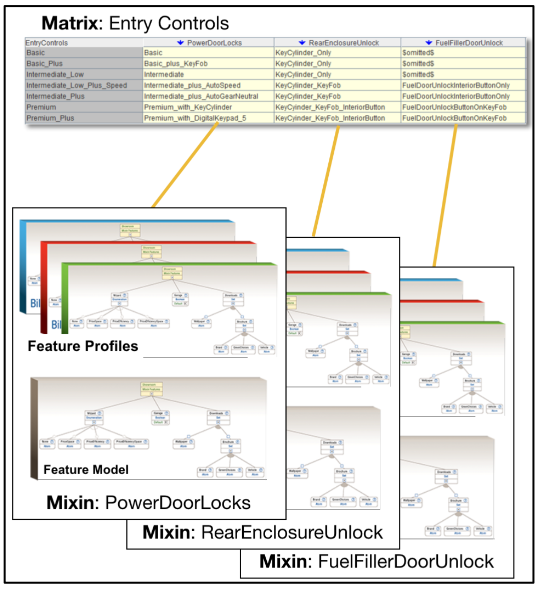


Figure 3: Feature Profile

Production Lines to Combine Multiple Mixins



EntryControls	PowerDoorLocks	RearEnclosureUnlock	FuelFillerDoorUnlock
Basic	Basic	KeyCylinder_Only	$omitted$
Basic_Plus	Basic_plus_KeyFob	KeyCylinder_Only	$omitted$
Intermediate_Low	Intermediate	KeyCylinder_Only	$omitted$
Intermediate_Low_Plus_Speed	Intermediate_plus_AutoSpeed	KeyCylinder_KeyFob	FuelDoorUnlockInteriorButtonOnly
Intermediate_Plus	Intermediate_plus_AutoGearNeutral	KeyCylinder_KeyFob	FuelDoorUnlockInteriorButtonOnly
Premium	Premium_with_KeyCylinder	KeyCyinder_KeyFob_InteriorButton	FuelDoorUnlockButtonOnKeyFob
Premium_Plus	Premium_with_DigitalKeypad_5	KeyCyinder_KeyFob_InteriorButton	FuelDoorUnlockButtonOnKeyFob

After primitive feature models are partitioned into mixins, the result is a collection of mixins that can still be quite large – possibly thousands of mixins in a large product line. Furthermore, considering the ultimate objective of the mixin, there needs to be an ontological element that combines one or more mixins and the shared asset supersets that are configured, so that the PLE Factory Configurator can apply the former to the latter. We call this encapsulation for one or more mixins and assets a *production line*. In the same way that mixins partition a broad primitive feature model into modular sub-models, production lines partition a broad collection of higher-order mixin features into modular sub-models.

Defining the scope of a production line is, like scoping a mixin, also an architectural activity. A production line will often represent a subsystem that is part of the products of the product line, with the shared assets associated with that subsystem.

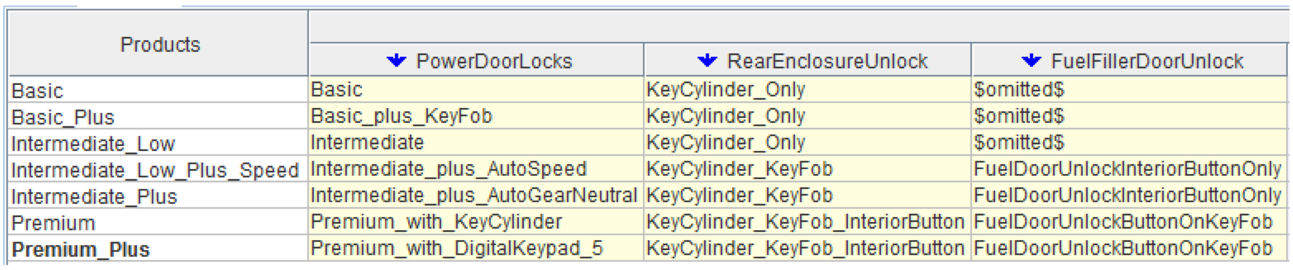
Each production line has in it the set of mixins that contain the features needed to configure the variation points in those assets. If the mixin is within the scope of the production line, it can be said to be *owned* by that production line. But sometimes a production line needs to refer to other mixins in order to configure an asset. In that case, it can *import* a mixin (by reference) from the owning production line. A mixin can be owned by one production line, but imported into multiple production lines if its features are crosscutting. Imported mixins are used to manage feature dependencies and constraints across subsystems.

Production Line Profiles and the Production Line Matrix

A production line typically contains multiple mixins. Each mixin is a higher-order feature with its mixin profile list enumerating the choices available for that mixin. Thus, in order to instantiate a production line configuration (and to configure a production line's shared assets) we need to specify a profile choice for each mixin in the production line. For example, if our production line contains three mixins, we need to make a feature profile choice for each of these three to create *a production line profile* and record that combination of choices. In Gears we record these choices for a production line profile in a row in a table called a *production line composition matrix*. Our matrix will have one column for each mixin in the production line, where we record the profile choice we make for that mixin.

To create multiple production line profiles, multiple rows are added to the production line matrix. Each profile row comprises a name and the set of choices we make for each of our production line's mixins.

Figure 4 shows the matrix from our EntryControls production line. It has seven production line profile rows recording the seven combinations of choices we make across the three mixins.



Products	PowerDoorLocks	RearEnclosureUnlock	FuelFillerDoorUnlock
Basic	Basic	KeyCylinder_Only	$omitted$
Basic_Plus	Basic_plus_KeyFob	KeyCylinder_Only	$omitted$
Intermediate_Low	Intermediate	KeyCylinder_Only	$omitted$
Intermediate_Low_Plus_Speed	Intermediate_plus_AutoSpeed	KeyCylinder_KeyFob	FuelDoorUnlockInteriorButtonOnly
Intermediate_Plus	Intermediate_plus_AutoGearNeutral	KeyCylinder_KeyFob	FuelDoorUnlockInteriorButtonOnly
Premium	Premium_with_KeyCylinder	KeyCylinder_KeyFob_InteriorButton	FuelDoorUnlockButtonOnKeyFob
Premium_Plus	Premium_with_DigitalKeypad_5	KeyCylinder_KeyFob_InteriorButton	FuelDoorUnlockButtonOnKeyFob

Figure 4: Production Line Profiles in a Production Line Matrix

Production Line as a Higher-Order Feature Abstraction

The production line and the list of named production line profiles in the production line matrix represent a higher order feature abstraction in our feature ontology. A named production line (Entry Controls in our example) now becomes a single higher-order feature that encapsulates and hides the low level details in its collection of mixins and mixin profile choices. The each named production line becomes a single higher-order feature and the list of named production line profiles become a higher-order enumeration of mutually exclusive choices for the higher-order production line feature. For example, our Entry Controls higher-order production line feature in Figure 4 has seven mutually exclusive profile choices, denoted by the row names in the first column.

It is not unusual to see a production line with millions of possible combinations of profile combinations having ten or less profile rows, thus once again reducing the available choices by orders of magnitude.

The roles responsible for engineering the production line are typically systems engineering experts for a subsystem within a product. They may collaborate with the portfolio planning organization to determine which differentiating characteristics should be offered as higher-order feature choices. The list of production line profiles that should be offered of also determined and defined by the systems engineering experts collaborating with the business to determine which offerings provide the best economic return for the enterprise.

Modularizing Multiple Production Lines into a Production Line of Production Lines

After all of the mixins are partitioned into production lines, the result is a collection of production lines that can still be quite large – possibly hundreds or even thousands of production lines in a large product line, as may be found in the automotive industry (Wozniak and Clements 2015). Since production lines act as higher-order features in the ontology, we can extend the definition of a production line to also include other production lines, resulting in a hierarchy, or tree (see Figure 5). In this way, we can mirror a system-of-systems architecture with a *production-line-of-production-lines hierarchy*. The constituent production lines provide places to make feature decisions corresponding to that part of the system-of-systems structure.

Defining the production line hierarchy is much like scoping an individual production line or scoping a mixin – it is an architectural activity. In this case the architectural objective is aligning it to a system-of-systems structure. The production line hierarchy can go to any level of depth. Each level in the hierarchy represents a new level of abstraction, where the production line becomes a new higher-order feature and it production line profiles (row in the production line matrix) become the available choices for that higher order features. Each level in the production line abstraction hierarchy often provides orders of magnitude reduction in the combinatoric complexity by limiting the number of specified profiles relative to the number of possible profiles.

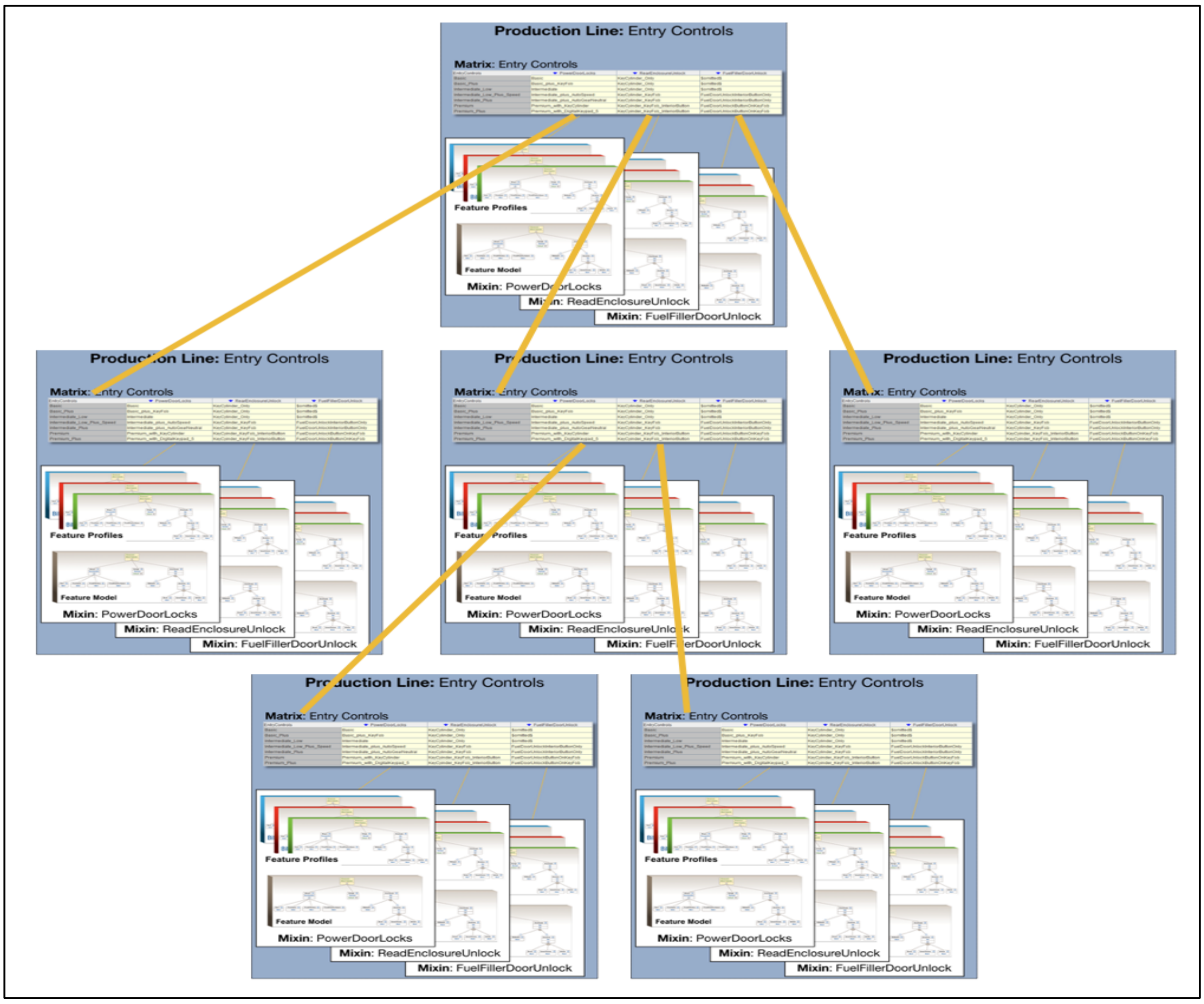

Production Line: Entry Controls
Matrix: Entry Controls
Feature Profiles
Feature Model
Mixin: PowerDoorLocks
Mixin: ReadEnclosureUnlock
Mixin: FuelFillerDoorUnlock

Figure 5: Production Line of Production Lines. An imported production line appears as a column in the matrix of its parent, the importing production line. Rows in the imported production line become available as cell choices in that column.

Product Family Tree

In product lines with a large number of members, products often occur in "clusters," meaning that some products have more in common with each other in terms of feature choices than they do with products outside that cluster. In the automotive realm, for example, SUVs have more in common with each other than they do with sports cars, although both are members of the same product line.

The commonality manifests itself as shared feature choices. SUVs have four doors, big engines with towing capacity, a large cargo space, etc. Sports cars have two doors, powerful engines that are optimized for acceleration rather than hauling, small (or no) cargo space, etc. In other words, products in a cluster have significant numbers of choices in common.

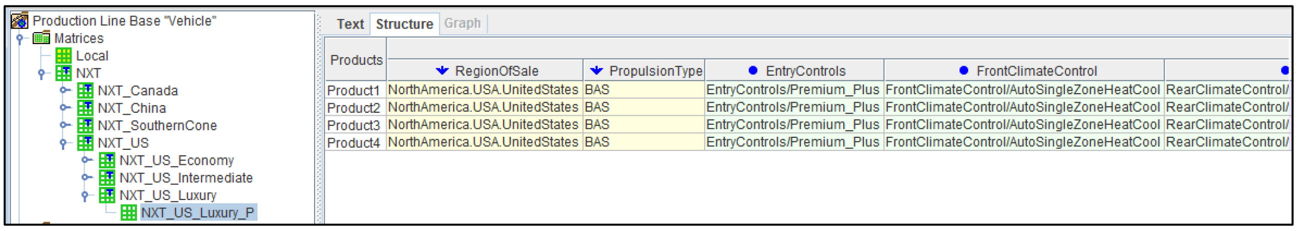
A cluster may itself consist of sub-clusters, and so forth. The result is a product family tree (Krueger 2013); we can represent this with a tree of production line matrices. In the matrix that corresponds to a cluster, you can make all of the choices that are common across all of the products in that cluster. That is, you can fill in the matrix cells with the profile selections you choose. For an SUV cluster, you would fill in the SUV matrix with four doors, big engine, etc. You can leave other choices (cells in the matrix) unbound, to be filled in by a descendant of that matrix in the tree.

In Gears, you can leave a choice totally unbound – make no statement about its eventual value – or partially unbound through down-selection. Down-selection doesn't make a final choice, but removes some choices from consideration. For an SUV, we may not know exactly which big engine an individual SUV will have, but we can rule out many that we know it won't have.

Matrix family trees significantly reduce the task of defining products, because at the lower levels of the tree (where products eventually are defined) many of the choices have already been made or narrowed. This also represents orders of magnitude reduction in the available combinations at each level in the family tree.

Matrix family trees also invite a separation of labor. Portfolio managers for a cluster can make cluster-wide decisions, and mangers for individual products can fill in the rest.

Figure 6 shows a family tree for a hypothetical vehicle product line called "NXT". NXT has four sub-families, defined by the region in which the vehicles will be sold. Sub-family "NXT_US" has itself three sub-families, corresponding to trim levels (economy, intermediate, luxury). The NXT US Luxury family (highlighted in the browser and listed on the right) has four members.



Production Line Base "Vehicle"
- Matrices
  - Local
  - NXT
    - NXT_Canada
    - NXT_China
    - NXT_SouthernCone
    - NXT_US
      - NXT_US_Economy
      - NXT_US_Intermediate
      - NXT_US_Luxury
        - NXT_US_Luxury_P

Products	RegionOfSale	PropulsionType	EntryControls	FrontClimateControl	
Product1	NorthAmerica.USA.UnitedStates	BAS	EntryControls/Premium_Plus	FrontClimateControl/AutoSingleZoneHeatCool	RearClimateControl/
Product2	NorthAmerica.USA.UnitedStates	BAS	EntryControls/Premium_Plus	FrontClimateControl/AutoSingleZoneHeatCool	RearClimateControl/
Product3	NorthAmerica.USA.UnitedStates	BAS	EntryControls/Premium_Plus	FrontClimateControl/AutoSingleZoneHeatCool	RearClimateControl/
Product4	NorthAmerica.USA.UnitedStates	BAS	EntryControls/Premium_Plus	FrontClimateControl/AutoSingleZoneHeatCool	RearClimateControl/

Figure 6: Matrix family tree

Matrix Proliferation Bundles

At this point in our ontology, we are down to defining a product by making a number of choices that we find numbers in the dozens, say 50 or so. For very large production lines, the number might be in the hundreds, but 50 is a nominal number to consider going forward.

Organizations that use product family trees to specify thousands or tens of thousands of products don't want to painstakingly fill out a matrix for every one of them. Modern PLE Factory Configurators have a "proliferate" feature that can automatically produce rows in a matrix at the leaf of the hierarchy for every combination of profile choices that are still unbound. A proliferated matrix lays out all the choices for all the products that we can now, for example, take to manufacturing.

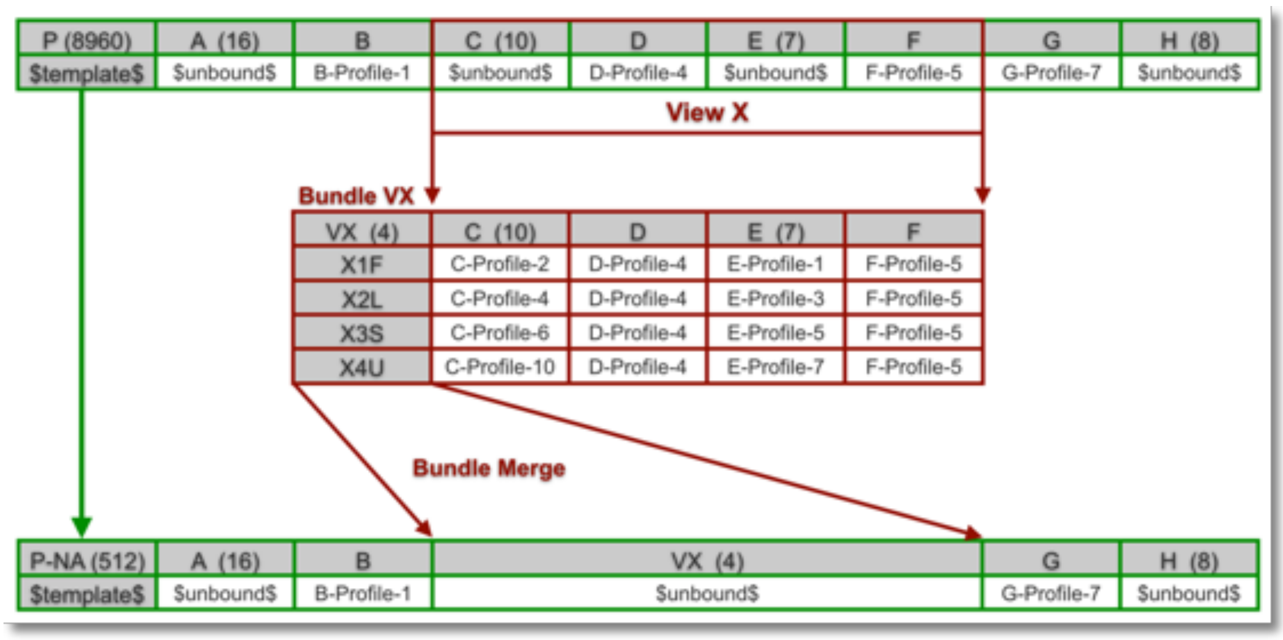
However, 50 or so choices still leaves us with 2^{50} possible combinations of choices, which means we could define 2^{17} (about 131,000) absolutely unique products for every one of the 2^{33} people on Earth. Before we push the "proliferate" button, we need a way to further pare down the choices.

In the same way that feature profiles reduced the potential complexity inherent in primitive feature models, we will use *matrix proliferation bundles* to whittle down the potential choices. The idea is to create meaningful bundles of the unbound selections to limit proliferation.

Figure 7 illustrates the concept. At the top is a template matrix with eight columns A-H. We have annotated the number of choices that remain available in each column: There are 16 choices available for A, 10 for C, 7 for E, and 8 for H. All other columns have fully bound selections. At this point we have the possibility of 16 x 10 x 7 x 8 = 8960 unique products. This is too many; this wide variety would overwhelm our manufacturing capability.

We realize that we don't need or want the 70 combinations possible from columns C through F. Instead, we decide that we create a bundle called VX that offers 4 combinations instead. By defining that bundle and letting subsequent (descendant) matrices choose from its choices, we have slashed our possible number of products to 512.

These manufacturing bundles, in which combinations of different features are limited and offered as packages, are common in the automotive industry. General Motors calls them "Regular Production Options;" other auto-makers have their own terminology. These so-called "RPO codes" comprise feature bundles that are available to a customer, such as a Sports package that combines a high-performance engine, a particular transmission, a tight suspension system, a prescribed steering wheel cover, external paint trim, and more.



P (8960)	A (16)	B	C (10)	D	E (7)	F	G	H (8)
$template$	$unbound$	B-Profile-1	$unbound$	D-Profile-4	$unbound$	F-Profile-5	G-Profile-7	$unbound$

VX (4)	C (10)	D	E (7)	F
X1F	C-Profile-2	D-Profile-4	E-Profile-1	F-Profile-5
X2L	C-Profile-4	D-Profile-4	E-Profile-3	F-Profile-5
X3S	C-Profile-6	D-Profile-4	E-Profile-5	F-Profile-5
X4U	C-Profile-10	D-Profile-4	E-Profile-7	F-Profile-5

P-NA (512)	A (16)	B	VX (4)	G	H (8)
$template$	$unbound$	B-Profile-1	$unbound$	G-Profile-7	$unbound$

Figure 7: Matrix Proliferation Bundles

Matrix proliferation bundles have, for very large product lines, finally brought us down into a decision space that is manageable, understandable, and manufacture-able.

Summary and Conclusions

Among the more powerful engineering techniques for reducing cognitive complexity for engineers in any discipline are abstraction and automation: Complex constructs are encapsulated and hidden behind simplifying abstractions, while automation provides a mapping from the abstractions to the hidden and more complex engineering constructs. Layers in the feature ontology serve exactly this purpose. Features in the Feature Catalog, at whatever ontological layer they appear, provide a simplifying abstraction that hides the exponential complexity of possibly hundreds, thousands or even millions of interacting points of variation within the engineering assets.

The concept of feature allows a consistent abstraction to be employed when defining and making choices, from a whole product configuration all the way down to the deployment of components within a low-level subsystem. Features provide the common communication vehicle - a *lingua franca* - among all stakeholders in the product line, from requirements engineers to testers, from marketers to executives, from designers to customers. Our ontology, summarized, is:

- Variation points in shared assets are configured by defined sets of choices (feature profiles) from primitive features.
- Features and profiles are packaged into mixins to achieve modularity.
- Production lines are mini-factories that contain related mixins and assets that they configure.
- Production lines can be structured into a hierarchy using an *imports* relation, enabling a system of systems to be represented as production line of production lines.
- Production lines contain a matrix that defines offerings that comprise choices from among the profiles of the contained mixins and the imported production lines.
- Matrices can be structured into hierarchies, allowing for the representation of product family trees, with decisions common to members of a sub-family made in one place.
- Choices at the leaves of a product family tree can be packaged into bundles.

Element	Purpose	Roles that Utilize	Potential Combinatoric Complexity
Production Line Proliferation Bundles	Sales Feature Options. Constrain proliferation.	Sales Configuration Engineers	2^{10} - 2^{20}
Production Line Family Tree	Partially bind and down-select offered Matrix Profile options for a Product Family. Proliferation	Product Portfolio Design Engineers	2^{50}
Production Line Profile	Provide desired offerings of a system or subsystem product line.	Product Marketing, Product Portfolio, System and Subsystem Domain Engineers.	2^{100}
Production Line	Compose and scope Mixins. Hierarchical Product Line of Product Lines.	Systems Engineering, Product Line Architects	2^{1000}
Feature Profile	Provide the desired offerings of a Mixin	System/Subsystem Product Line Design Engineers	2^{1000}
Feature Mixin	Modularize and scope related features	Feature Architects	$2^{10,000}$
Primitive Feature	Root-cause Feature Abstraction. functional, nonfunctional, and deployment variation.	System/Subsystem Feature Designers	$2^{10,000}$
Shared Asset Variation Point	Feature-Based asset variation management	Asset Engineers	$2^{1,000,000}$

Figure 8: How the Enterprise Feature Ontology Leads to Manageable Complexity

Figure 8 summarizes our enterprise feature ontology, and shows the kinds of roles involved in decision-making and selection at each level. At each step along the way, the number of choices available to engineers and other enterprise stakeholders working at that level shrinks by many orders of magnitude. A decision space on the order of $2^{1,000,000}$ (at the shared asset variation point level) becomes a decision space on the order of 2^{10}-2^{20}.

For smaller production lines with, say, dozens of product instances, the ontology will produce tractable decision spaces much earlier. In that case, a few of the ontology's lower layers (e.g., features, profiles, and mixins, packaged into a single production line) can suffice; however, in practice, we observe that most product lines in this size still avail themselves of the separation of concerns brought about by the production-line-of-production-lines structure.

We began with a question: How do you bridge the gap between potentially1,000,000 variation points in a product line's shared assets down to a few dozen customer-facing decisions? Our feature ontology is the answer. This ontology came about through practice and experience, not speculation or imagination. As Feature-Based PLE made larger and larger strides into larger and larger product lines, each layer in the model was added on top of previous layers as a result of need. At each level, we combined large numbers of available choices into smaller numbers of pre-packaged selections. There is no reason the ontology could not be extensible in this way by adding still more layers to the top, should the need arise to work with product lines orders of magnitude larger than the largest ones today. We look forward to seeing Feature-Based PLE applied in those settings.

References

BigLever Software, "BigLever Software Gears," http://www.biglever.com/solution/product.html, downloaded April 2017.

Brownsword, L., and Clements, P., "A Case Study in Successful Product Line Development" (CMU/SEI-96-TR-016, ADA315802). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. http://www.sei.cmu.edu/publications/documents /96.reports/96.tr.016.html.

Clements, P., Gregg, S., Krueger, C., Lanman, J., Rivera, J., Scharadin, R., Shepherd, J., and Winkler, A., "Second Generation Product Line Engineering Takes Hold in the DoD," Crosstalk, The Journal of Defense Software Engineering, USAF Software Technology Support Center, 2013.

Clements, P.; Northrop, L. *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.

Dillon, M., Rivera, J., Darbin, R., Clinger, B., "Maximizing U.S. Army Return on Investment Utilizing Software Product-Line Approach," Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC), 2012.

Flores, R., Krueger, C., Clements, P. "Mega-Scale Product Line Engineering at General Motors," Proceedings of the 2012 Software Product Line Conference (SPLC), Salvador Brazil, August 2012.

Gregg, S, Scharadin, R., Clements, P. "The More You Do, the More You Save: The Superlinear Cost Avoidance Effect of Systems and Software Product Line Engineering, Proceedings Software Product Line Conference 2015, Nashville, 2015.

Gregg, S., Scharadin, R., LeGore, E., Clements, P. "Lessons from AEGIS: Organizational and Governance Aspects of a Major Product Line in a Multi-Program Environment," Proceedings, Software Product Line Conference 2014, Florence, Italy, 2014.

Gruber, T.R. (1993). A Translation Approach to Portable Ontology Specification. Knowledge Acquisition 5: 199-220.

Kang, K.; Cohen, S.; Hess, J.; Novak, W.; & Peterson, A. "Feature-Oriented Domain Analysis (FODA) Feasibility Study" (CMU/SEI-90-TR-021, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

Krueger, C. “Multistage Configuration Trees for Managing Product Family Trees” Proceedings of the 2013 Software Product Line Conference (SPLC), Tokyo, Japan, August 2013.

Krueger, C. and Clements, P. “Systems and Software Product Line Engineering,” Encyclopedia of Software Engineering, Philip A. LaPlante ed., Taylor and Francis, 2013, in publication.

INCOSE Product Line Engineering International Working Group, http://www.incose.org/ChaptersGroups/WorkingGroups/analytic/product-lines, downloaded 09 November 2016.

Linden, Frank J. van der, Schmid, Klaus, Rommes, Eelco. Software Product Lines in Action, Springer, 2007.

Noy, Natalya F., and McGuinness, Deborah L., “Ontology Development 101: A Guide to Creating Your First Ontology,” Stanford University, Stanford University, Stanford, CA, 94305,http://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html, downloaded 09 November 2016.

Parnas, David L., “On the criteria to be used in decomposing systems into modules,” Communications of the ACM, Volume 15 Issue 12, pp. 1053-1058, Dec. 1972.

Saitta, Lorenza, and Zucker, Jean-Daniel, Abstraction in Artificial Intelligence and Complex Systems, Springer Science & Business Media, 2013.

Software Engineering Institute, “Benefits and Costs of a Product Line,” http://www.sei.cmu.edu/productlines/frame_report/benefits.costs.htm, downloaded April 2017

Software Engineering Institute, “Catalog of Software Product Lines,” http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm, downloaded April 2017

SPLC Product Line Hall of Fame, http://splc.net/fame.html, downloaded April 2017

Villanueva, J. C., “Atoms in the Universe,” http://www.universetoday.com/36302/atoms-in-the-universe, 2009.

Wozniak, L., Clements, P. “How Automotive Engineering Is Taking Product Line Engineering to the Extreme,” *Proc. SPLC 2015*, Nashville, 2015.

Biography

Dr. Charles Krueger is founder and CEO of BigLever Software, Inc., the industry’s leading provider of product line engineering technology and services. An acknowledged thought leader in the PLE field, he has more than 25 years of experience in software engineering practice and more than 60 articles, columns, book chapters, conference keynotes and session presentations. Through active involvement in key industry events and organizations, he brings innovative PLE concepts, new generation methodologies, and success stories to the forefront of the systems and software engineering community. Dr. Krueger has played an instrumental role in establishing some of the industry’s most notable PLE practices at leading companies in automotive, aerospace and defense, aviation systems, alternative energy, e-commerce, and computer systems. He received his PhD in computer science from Carnegie Mellon University.

Dr. Paul Clements is the Vice President of Customer Success at BigLever Software, Inc., where he works to spread the adoption of systems and software product line engineering. He was previously at Carnegie Mellon’s Software Engineering Institute, where for 17 years he worked in software product line engineering and software architecture documentation and analysis. Clements is co-author of three practitioner-oriented books about software architecture as well as the field’s leading text on software product line engineering. His PhD in computer sciences is from the University of Texas at Austin.